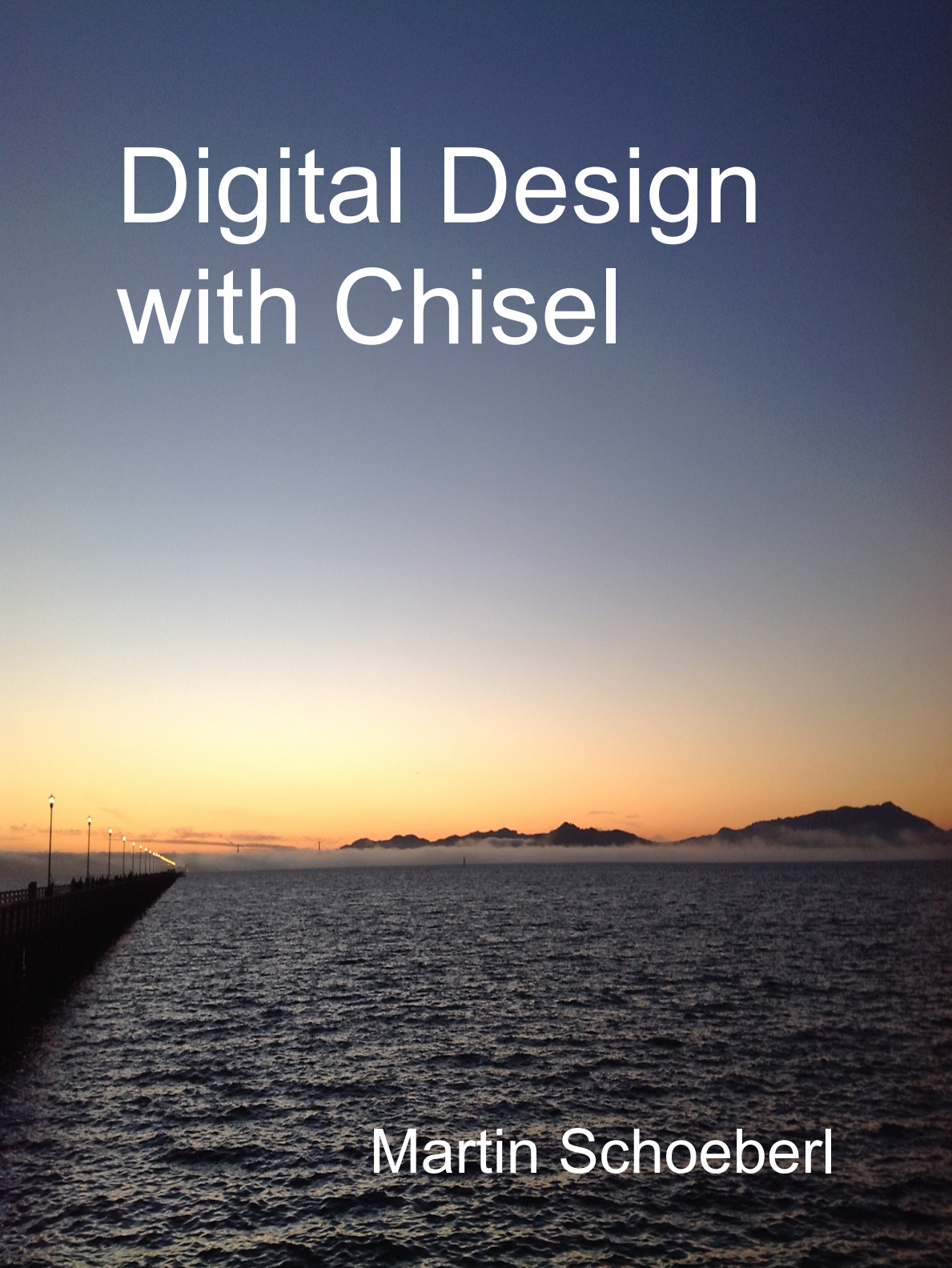
Digital Design with Chisel



Martin Schoeberl

Chisel 로하는디지털설계

제 7 판

Chisel 로하는디지털설계

제 7 판

Martin Schoeberl

이메일: martin@jopdesign.com
소스는 https://github.com/schoeberl/chisel-book에서확인하십시오

초판은 2019 년 Kindle Direct Publishing 에서출간되었습니다,
https://kdp.amazon.com/

미국의회도서관도서목록자료

Schoeberl, Martin

Digital Design with Chisel
Martin Schoeberl
Includes bibliographical references and an index.
ISBN 9781689336031

미국에서제작되었습니다.
Martin Schoeberl 이조판했습니다.

Contents

List of Figures

List of Tables

Listings

서문

디지털설계의세계에몸담기에지금은흥미진진한시기입니다. 데나드스케일링의종말과무어의법칙둔화로인해, 이분야에서혁신이이보다더절실했던적은아마없을것입니다. 반도체기업들은얻을수있는모든성능을계속해서짜내고있지만, 이러한개선의비용은급격히상승해왔습니다. Chisel 은생산성을향상시켜이비용을줄여줍니다. 설계자가더적은시간에더많이만들수있고, 재사용을통해검증비용을분산시킬수있다면, 기업은비반복엔지니어링 (NRE) 비용을줄일수있습니다. 또한학생과개인기여자모두스스로더쉽게혁신할수있습니다.

Chisel 은또다른프로그래밍언어인 Scala 에내장되어있다는점에서대부분의언어와다릅니다. 근본적으로 Chisel 은동기식디지털회로를표현하는데필요한기본요소들을나타내는클래스와함수의라이브러리입니다. Chisel 설계는사실상실행되면서회로를 생성하는 Scala 프로그램입니다. 많은사람들에게이는직관에어긋나는것처럼보일수있습니다: ” 왜 Chisel 을 VHDL 이나 SystemVerilog 처럼독립적인언어로만들지않았는가?” 이질문에대한제답은다음과같습니다: 소프트웨어세계는지난수십년동안설계방법론에서상당한혁신을이루어왔습니다. 이러한기법을새로운하드웨어언어에적응시키려하기보다는, 우리는그저현대적인프로그래밍언어를 사용하여그이점을공짜로얻을수있습니다.

Chisel 에대한오랜비판중하나는” 배우기어렵다” 는것입니다. 이러한인식의상당부분은전문가들이자신의연구나상업적필요를해결하기위해만든크고복잡한설계가널리퍼져있기때문입니다. C++ 와같은대중적인언어를배울때, GCC 의소스코드를읽는것부터시작하지는않습니다. 오히려신규학습자를위한수많은강좌, 교재, 그리고기타학습자료가존재합니다. *Chisel* 를활용한디지털설계 (Chisel 을활용한디지털설계) 에서 Martin 은 Chisel 을배우고자하는모든사람을위한중요한자료를만들었습니다.

Martin 은경험이풍부한교육자이며, 이는이책의구성에그대로드러납니다. 설치와기본요소부터시작하여, 그는건물을짓듯이벽돌한장한장씩독자의이해를쌓아올립니다. 포함된연습문제는이해를굳히는모르타르로서, 각개념이독자의마음속에자리잡도록보장합니다. 이책은나머지구조에목적을부여하는지붕처럼하드웨어생성기로절정에이릅니다. 마지막에이르면, 독자는단순하지만유용한설계인 RISC 프로세서를만들수있는지식을갖추게됩니다.

Chisel 를활용한디지털설계에서 Martin 은생산적인디지털설계를위한튼튼한토대를마련했습니다. 그것으로무엇을만들지는여러분에게달려있습니다.

Jack Koenig
Chisel and FIRRTL Maintainer
Staff Engineer, SiFive

머리말

이책은하드웨어구성언어 Chisel 의사용에초점을맞춘디지털설계입문서입니다. Chisel 은객체지향언어와함수형언어같은소프트웨어공학의발전을디지털설계에접목합니다.

이책은하드웨어설계자와소프트웨어엔지니어를대상으로합니다. Verilog 나 VHDL 에대한지식을갖춘하드웨어설계자는최신언어를사용하여다음 ASIC 또는 FPGA 설계의생산성을향상시킬수있습니다. 객체지향프로그래밍과함수형프로그래밍에대한지식을갖춘소프트웨어엔지니어는자신의지식을활용하여하드웨어를프로그래밍할수있으며, 예를들어클라우드에서실행되는 FPGA 가속기가그예입니다.

이책의접근방식은작거나중간크기의전형적인하드웨어컴포넌트를제시하여 Chisel 을이용한디지털설계를탐구하는것입니다.

저는단한문장도대형언어모델 (LLM) 을사용하여작성하지않았습니다. 모든 실수는 LLM 의환각이아니라제자신의것입니다. 문법검사에는 Grammarly 를, 코드작성에는 Copilot 을사용합니다.

2 판서문

Chisel 이애자일하드웨어설계를가능하게하듯이, 오픈액세스와주문형인쇄도 애자일교재출판을가능하게합니다. 이책의초판이나온지 6 개월도채지나지않아개선되고확장된 2 판을제공할수있게되었습니다.

사소한수정외에 2 판의주요변경사항은다음과같습니다. 테스트관련절이확장되었습니다. 순차빌딩블록장에는더많은예제회로가포함되었습니다. 입력처리에관한새로운장에서는입력동기화를설명하고, 디바운싱회로를설계하는방법과잡음이섞인입력신호를필터링하는방법을보여줍니다. 예제설계장은 FIFO 의여러구현을보여주도록확장되었습니다. FIFO 변형은또한디지털설계에서타입매개변수와상속을사용하는방법을보여줍니다.

3 판서문

Chisel 은지난 1 년간계속발전해왔으므로, Chisel 책의새판을낼때가되었습니다. 모든예제를최신버전의 Chisel(3.5.3) 로변경하였고, 권장 Scala 버전(2.12.13) 도변경하였습니다.

Chisel 3.5 에서는 iotesters 패키지의일부인테스트환경 PeekPokeTester 가지원중단되었습니다. 따라서테스트관련설명을새로운 ChiselTest 프레임워크로변경하였습니다. PeekPokeTester 를사용하는 Chisel 설계가여전히많이존재하므로, 이에대한설명은부록으로옮겼습니다.

Chisel/Scala/Java 환경의매력적인측면중하나는오픈소스라이브러리를배포하기위한기존인프라에편승할수있다는점입니다. 다른오픈소스 Java 라이브러리와마찬가지로간단하게하드웨어컴포넌트를 Maven 에게시할수있습니다. Maven 에게시한다는것은 build.sbt 설정에단하나의참조만추가하면서드파티컴포넌트를컴파일흐름에통합할수있다는의미입니다. 이는설계에 chisel 라이브러리를포함하는방식과동일한과정입니다. Chisel 설계를 Maven Central 에게시하는방법에관한절을추가하였습니다.

더간단한예제를통해컴포넌트에대한설명을개선하였습니다.

하드웨어 생성기는 Scala 로작성됩니다. 따라서 Scala 에관한짧은절을추가하였습니다. 하드웨어생성기장에는함수형프로그래밍을이용하여생성기를작성하는방법에관한절을추가하여확장하였습니다.

부록에는예약어목록과약어목록이추가되어확장되었습니다.

Hans Jakob Damsgaard 는 Chisel 의 BlackBox 를통해외부컴포넌트를사용하는방법과클록도메인교차 (다중클록메모리) 를위해메모리를사용하는방법에관한설명을기고하였습니다.

4 판서문

4 판에서는실제 Chisel 버전 3.5.4 로전환하였습니다. 조합빌딩블록장에아비터, 우선순위인코더, 비교기를추가하였습니다. 하드웨어생성장은간단한 2 대 1 중재회로로부터공정한중재트리를구성하는것을포함하여더많은함수형예제로확장하였습니다. 인터커넥트, 버스인터페이스, 그리고 IO 장치를메모리매핑장치로연결하는방법에관한새로운장을추가하였습니다. 디버깅, 테스트, 검증에관한새로운장을시작하였습니다. 이중요한주제에관한장은다음판에서확장할계획입니다. 프로세서장은데이터패스그림을포함하여마이크로프로세서에대한보다완만한소개로확장하였습니다.

5 판서문

5 판에서는실제 Chisel 버전 3.5.6 과 Scala 2.13 으로업그레이드하였습니다. 더고급주제를위한여지를만들기위해 Chisel 2 와 PeekPokeTester 에관한두부록을삭제하였습니다. 현재활발히유지관리되고있는모든프로젝트는최소한호환성레이어를통해서라도결국 Chisel 3 으로이전하였습니다. PeekPokeTester 는 Chisel 3.5 부터지원중단되었으며, ChiselTest 로전환할것을권장합니다. 필요하다면, 이두장은이책의 웹페이지에 PDF 로제공되는 Chisel 책의이전판에서확인하실수있습니다.

5 판에는큰변경사항이포함되어있지않으며, 이는정리판에해당합니다. 문장을더명료하게다듬기위해상당한교정작업을수행하였습니다. Chisel 은이판에서다루는작은편의기능 (예: ChiselEnum) 을추가하였습니다. 프로세서장을확장하였고 Leros 코드스니펫을실제 Leros 버전에맞게업데이트하였습니다.

6 판서문

Chisel 은최근몇가지주요변화를겪었습니다. Chisel 3.x 까지는 Chisel 이주로 Scala 를기반으로하였는데, 이는 Java 가상머신에서실행되어플랫폼에독립적이라는것을의미합니다. 3.x 이후에는 Chisel 컴파일파이프라인의아키텍처가변경되었습니다. 이러한변화를나타내기위해버전번호체계가변경되어메이저 버전번호가더자주증가하게되었습니다. 4.0 버전은건너뛰었으며, 다음 Chisel 버전은 5.0 입니다. 5.0 버전부터컴파일러백엔드는 LLVM 프로젝트의일부인 회로컴파일러 CIRCT를기반으로합니다. Chisel 사용자에게있어주된차이점은 CIRCT(및 LLVM) 가 C++ 로작성되어있어운영체제 (프로세서아키텍처) 마다다른실행파일이필요하다는것입니다.

이책은 Chisel 버전 3.5.6, 3.6.1, 5.3.0, 6.5.0 으로테스트하였습니다. Chisel 3.6 은 Verilog 파일을생성하기위해 Scala 기반백엔드를사용하는마지막버전입니다. Chisel 5 는 CIRCT 를사용하며 firtool 을수동으로설치해야합니다. Chisel 6 의최신버전은 firtool 바이너리를포함합니다. ChiselTest 를이용한테스트는 Chisel 6 까지는여전히지원됩니다. 따라서 Chisel 5 는건너뛰고 Chisel 6 을사용할것을권장합니다. Chisel 6 은다른시뮬레이션및테스트라이브러리로전환할수도있습니다. 저희는 Scala 2.13 을사용하였고 Java 17 과 Java 21 로테스트하였습니다. Chisel 3.5.6 은 Java 21 을지원하지않습니다. 따라서 Chisel 3 을사용하고자한다면 Chisel 3.6.1 을사용할것을권장합니다.

TODO: 이중일부는시작하기부분에포함될수도있습니다.

TODO: Leros 코드도더보여주어야합니다.

저희는테스팅장에내부신호에접근하는방법과어서션을사용하는방법에대한설명을추가했습니다. 또한 Chisel 에서의정형검증도탐구하기시작했습니다.

성능을높이기위해디지털회로를파이프라인으로조직하는것은디지털설계와컴퓨터구조에서핵심적인측면입니다. 저희는 3 단계 RISC-V 파이프라인을예제로사용하는파이프라이닝장을추가했습니다.

저희는 Chisel 이하드웨어설계에서여전히틈새언어이며, 최소한 VHDL 및/또는 Verilog 를읽을수있어야한다는점을잘알고있습니다. 그러나디지털설계에대한좋은지식을갖추고있다면, 다른하드웨어기술언어로전환하는것은며칠이면충분할것이라고생각합니다. 이러한전환을간소화하기위해, 저희는 Chisel, VHDL, Verilog 의기본구성요소를보여주고비교하는부록을추가했습니다.

이책은텍스트를작성하거나다시작성하는데인공지능이나대규모언어모델의도움을전혀받지않고집필되었습니다. 저희는문법검사와문체개선을위해 Grammarly[1]를사용했습니다.

제 7 판서문

번역본

이책은중국어, 일본어, 베트남어로번역되었습니다. 모든번역본은이책의 웹페이지에서무료 PDF 로이용하실수있습니다. 중국어번역을맡아주신 Yuda Wang, Qiwei Sun, Yun Chen 님, 일본어번역을맡아주신 Seiji Munetoh, Masatoshi Tanabata, Takaaki Hagino 님, 그리고베트남어번역을맡아주신 VieLe Duc Hung 님께감사드립니다. 만약이책을다른언어로번역하는데관심이있으시다면, 자유롭게번역하시고동일한라이선스로공개하십시오. 그런다음저에게연락해주시면, 제가여러분의번역본을안내해드릴수있을것입니다.

감사의말씀

이렇게멋진하드웨어구성언어를만들어주신 Chisel 작업에참여한모든분들께감사드립니다. Chisel 은사용하는것자체가매우즐거우며, 그렇기에책을쓸만한가치가있습니다. 저는매우우호적이고친절하며 Chisel 에관한질문에답하는데지치는법이없는 Chisel 커뮤니티전체에감사드립니다.

[1] https://app.grammarly.com/

또한최근몇년간고급컴퓨터구조과목에서대부분최종프로젝트를위해 Chisel 을익힌저의학생들에게도감사드립니다. 익숙한영역을벗어나최첨단하드웨어기술언어를배우고사용하는여정에나서주셔서감사합니다. 여러분의많은질문들이이책의모습을다듬는데도움이되었습니다.

2020 년부터덴마크공과대학교에서디지털전자공학을가르치며 Chisel 을사용한것은즐거운경험이었습니다. 2 학기에 Chisel 과 Java 를동시에익히는것이도전적인일임을저는알고있습니다. 하드웨어기술을위한현대적인프로그래밍언어를익히는데열린마음을가져준이과목의모든학생들에게감사드립니다.

제 3 판에서는, 이책의모든테스트코드를 ChiselTest 로다시작성하고, 테스팅장에 ChiselTest 를추가하고, 블랙박스설명과멀티클록메모리예제를추가해주신 Hans Jakob Damsgaard(@hansemandse) 님께감사드리고자합니다.

1 서론

이책은현대적인하드웨어구성언어인 Chisel [5] 를사용한디지털시스템설계입문서입니다. 이책에서저희는여러분이더짧은시간에더복잡하고상호작용하는디지털시스템을구축할수있도록, 일반적인디지털설계서적보다더높은추상화수준에초점을맞춥니다.

이책과 Chisel 은두부류의개발자, 즉 (1) 하드웨어설계자와 (2) 소프트웨어프로그래머를대상으로합니다. VHDL 이나 Verilog 에능숙하고하드웨어생성을위해 Python, Java, Tcl 과같은다른언어를사용하는하드웨어설계자는, 하드웨어생성이언어자체의일부인단일하드웨어구성언어로옮겨갈수있습니다. 소프트웨어프로그래머는, 예를들어 Intel 의미래칩이프로그램을가속화하기위한프로그래머블하드웨어를포함하게됨에따라하드웨어설계에관심을갖게될수있습니다. Chisel 을여러분의첫번째하드웨어기술언어로사용하는것은전혀문제가되지않습니다.

Chisel 은객체지향및함수형프로그래밍과같은소프트웨어공학의발전을디지털설계영역에가져옵니다. Chisel 은여러분이레지스터전송수준에서하드웨어를표현할수있게해줄뿐만아니라, 하드웨어 생성기를작성할수있게도해줍니다.

오늘날하드웨어는일반적으로하드웨어기술언어로기술됩니다. CAD 도구를사용하더라도하드웨어컴포넌트를그림으로그리던시대는끝났습니다. 일부고수준개략도는시스템에대한개요를제공할수는있지만, 시스템을기술하기위한용도는아닙니다. 가장흔한두가지하드웨어기술언어는 Verilog 와 VHDL 입니다. 두언어모두오래되었고, 많은유산을안고있으며, 언어의어떤구성요소가하드웨어로합성가능한지에대한경계가계속변화하고있습니다. 오해하지는마십시오: VHDL 과 Verilog 는 ASIC으로합성될수있는하드웨어블록을완벽하게기술할수있습니다. Chisel 을이용한하드웨어설계에서, Verilog 는테스팅과합성을위한중간언어역할을합니다.

이책은디지털설계와그기초에대한일반적인입문서가아닙니다. CMOS 트랜지스터로게이트를만드는방법과같은디지털설계의기초에대한입문은다른디지털설계서적을참고하시기바랍니다. 그러나이책은 ASIC 이나 FPGA를대상으로하는설계를기술하는현재의실무관행에해당하는추상화수준에서디지털설계를가르치고자합니다.[1] 이책의선수지식으로, 저희는 불대수와 이진수체계에대한

[1] 저자가대상기술로서 ASIC 보다 FPGA 에더익숙하기때문에, 이책에나오는일부설계최적화는

기본적인지식을가정합니다. 또한어떤프로그래밍언어든프로그래밍경험이어느정도있다고가정합니다. Verilog 나 VHDL 에대한지식은필요하지않습니다. Chisel 은디지털하드웨어를기술하기위한여러분의첫번째프로그래밍언어가될수있습니다. 예제의빌드과정이 sbt 와 make 를기반으로하므로, 명령줄인터페이스 (CLI, 터미널또는 Unix 셸이라고도함) 에대한기본지식이도움이될것입니다.

Chisel 자체는큰언어가아닙니다. 기본구성요소는 한페이지에담길정도이며, 며칠안에익힐수있습니다. 따라서이책역시큰책이아닙니다. Chisel 은많은유산을안고있는 VHDL 및 Verilog 보다분명작습니다. Chisel 의힘은그자체로표현력이풍부한언어인 Scala 안에 Chisel 이내장되어있다는데서나옵니다. Chisel 은" 쓸수록느는언어" [29] 라는 Scala 의특성을물려받았습니다. 그러나 Scala 는이책의주제가아닙니다. 저희는하드웨어설계자를위한 Scala 에관한짧은절을제공합니다. Odersky 등의교재 [29] 는 Scala 에대한일반적인입문을제공합니다. 이책은디지털설계와 Chisel 언어에대한튜토리얼이며, Chisel 언어레퍼런스도아니고완전한칩설계에관한책도아닙니다.

이책의모든코드예제는 GitHub 에서지속적통합 (CI) 을통해컴파일및테스트된완전한프로그램에서추출되었습니다. 따라서이코드에는어떠한구문오류도포함되어있지않을것입니다. 코드예제는이책의 GitHub 저장소에서이용하실수있습니다. 저희는 Chisel 코드를보여드리는것외에도, 유용한설계와좋은하드웨어기술스타일의원칙을보여드리고자노력했습니다.

이책은태블릿 (예: iPad) 이나노트북에서읽기에최적화되어있습니다. 저희는본문중에추가로읽을거리에대한링크를포함시켰으며, 주로 Wikipedia 문서로연결됩니다.

1.1 Chisel 및 FPGA 도구설치하기

Chisel 은 Scala 라이브러리이며, Chisel 과 Scala 를설치하는가장쉬운방법은 Scala 빌드도구인 sbt 를사용하는것입니다. Scala 자체는 Java JDK 8(또는 Java 21 까지의이후버전) 의설치에의존합니다. Oracle 이 Java 의라이선스를변경했기때문에, AdoptOpenJDK에서 OpenJDK 를설치하거나 SDKMAN을사용해 Java 설치를관리하는편이더쉬울수있습니다.

더자세한설정안내는 chisel-lab의 Setup.md에서확인하실수있습니다. 첫번째실습에서는기존 Chisel 프로젝트를 IntelliJ 에서여는방법을설명합니다.

FPGA 기술을대상으로합니다.

1.1.1 macOS

AdoptOpenJDK에서 Java OpenJDK 8(최대 21 까지) 을설치하십시오. Mac OS X 에서는패킷관리자 Homebrew로 sbt 와 git 을다음과같이설치할 수있습니다.

```
$ brew install sbt git
```

GTKWave와 IntelliJ(커뮤니티에디션) 를설치하십시오. 프로젝트를임포트할때는이전에설치한 JDK 를선택하십시오.

1.1.2 Linux/Ubuntu

다음과같이 Ubuntu 에 Java 와유용한도구들을설치하십시오.

```
$ sudo apt install openjdk-8-jdk git make gtkwave
```

Debian 기반인 Ubuntu 에서는프로그램이보통 Debian 파일 (.deb) 로설치됩니다. 그러나이글을쓰는시점기준으로 sbt 는바로설치가능한패키지형태로제공되지않습니다. 따라서설치과정이조금더복잡합니다. sbt download의안내를따르십시오.

1.1.3 Windows

AdoptOpenJDK에서 Java OpenJDK(8 부터최대 21 까지) 를설치하십시오. Chisel 과 Scala 는 Windows 에서도설치하여사용할수있습니다. GTKWave와 IntelliJ(커뮤니티에디션) 를설치하십시오. 프로젝트를임포트할때는이전에설치한 JDK 를선택하십시오. sbt 는 Windows 설치프로그램으로설치할수있습니다. Installing sbt on Windows를참고하십시오. git 클라이언트를설치하십시오.

1.1.4 FPGA 도구

FPGA 용하드웨어를빌드하려면합성도구가필요합니다. 양대 FPGA 벤더인 Altera(현재 Intel 소속회사)[2]와 AMD[3]는소형에서중형크기의 FPGA 를다룰수있는무료버전도구를제공합니다. 이러한중형 FPGA 는멀티코어 RISC 스타

[2] former Intel and former Altera
[3] former Xilinx

일프로세서를만들수있을만큼충분히큽니다. Intel 은 Quartus Prime Lite Edition을, AMD 는 Vivado Design Suite, WebPACK Edition을제공합니다. 두도구모두 Windows 와 Linux 용으로는제공되지만, macOS 용으로는제공되지않습니다.

F4PGA를사용하면이제특정 FPGA 에대해완전한오픈소스합성도구를사용할수있습니다.

1.2 Hello World

프로그래밍언어를다루는모든책은 *Hello World* 예제라고불리는최소한의예제로시작해야합니다. 다음코드는첫번째시도입니다.

```
object HelloScala extends App {
  println("Hello Chisel World!")
}
```

sbt 로이짧은프로그램을컴파일하고실행하면

```
$ sbt run
```

Hello World 프로그램에서기대되는출력으로이어집니다.

```
[info] Running HelloScala Hello Chisel World!
```

그러나이것이 Chisel 일까요? 이것이문자열을출력하기위해생성된하드웨어일까요? 아닙니다. 이것은순수한 Scala 코드이며하드웨어설계를대표하는 Hello World 프로그램이아닙니다.

1.3 Chisel Hello World

그렇다면하드웨어설계를위한 Hello World 프로그램에해당하는것은무엇일까요? 유용하면서도눈에보이는최소한의설계는무엇일까요? 점멸하는 LED 는 Hello World 의하드웨어 (또는임베디드소프트웨어) 버전입니다. LED 가깜빡인다면, 우리는더큰문제를해결할준비가된것입니다!

Listing 1.1은 Chisel 로기술된점멸 LED 를보여줍니다. 이코드예제의세부사항을이해하는것은중요하지않습니다. 이는이후장에서다룰것입니다. 다만회로는보통예컨대 50 MHz 와같은높은주파수로클록킹되며, 눈에보이는점멸을얻기위해서는 Hz 범위의타이밍을얻어낼카운터가필요하다는점만알아두십

```
class Hello extends Module {
  val io = IO(new Bundle {
    val led = Output(UInt(1.W))
  })
  val CNT_MAX = (50000000 / 2 - 1).U

  val cntReg = RegInit(0.U(32.W))
  val blkReg = RegInit(0.U(1.W))

  cntReg := cntReg + 1.U
  when(cntReg === CNT_MAX) {
    cntReg := 0.U
    blkReg := ~blkReg
  }
  io.led := blkReg
}
```

Listing 1.1: A hardware Hello World in Chisel

시오. 위예제에서는 0 부터 25000000-1 까지카운트한다음점멸신호를토글하고 (blkReg := blkReg) 카운터를다시시작합니다 (cntReg := 0.U). 이하드웨어는그러면 1 Hz 로 LED 를점멸시킵니다.

1.4 Chisel 을위한 IDE

이책은여러분이사용하는프로그래밍환경이나에디터에대해어떠한가정도하지않습니다. 명령줄에서 sbt 를사용하고원하는에디터를사용하는것만으로도기본사항을쉽게배울수있어야합니다. 다른책들의전통을따라, 셸/터미널/CLI 에입력해야하는모든명령어앞에는 $ 문자가붙는데, 이는직접입력하지않아야합니다. 예를들어, 현재폴더의파일들을나열하는 Unix ls 명령어는다음과같습니다.

```
$ ls
```

그렇지만컴파일러가백그라운드에서실행되는통합개발환경 (IDE) 은코딩속도를높여줄수있습니다. Chisel 은 Scala 라이브러리이므로, Scala 를지원하는모든 IDE 는 Chisel 을위한좋은 IDE 이기도합니다. IntelliJ, Visual

Studio Code, Eclipse에서는 build.sbt 에있는 sbt 프로젝트설정으로부터프로젝트를생성하는것이가능합니다.

IntelliJ 에서는 Scala 플러그인을설치해야합니다. 그런다음 *File - New - Project from Existing Sources...* 를통해기존소스로부터새프로젝트를생성하고, 프로젝트의 build.sbt 파일을선택하면됩니다.

Eclipse 에서는다음을통해프로젝트를생성할수있습니다

```
$ sbt eclipse
```

그리고해당프로젝트를 Eclipse 로임포트합니다.[4]

Visual Studio Code는 Chisel IDE 의또다른선택지입니다. Scala Metals 확장은 Scala 지원을제공합니다. 왼쪽바에서 *Extensions* 를선택하고 *Metals* 를검색하여 *Scala (Metals)* 를설치하십시오. sbt 기반프로젝트를임포트하려면 *File - Open* 으로폴더를여십시오.

1.5 소스접근과전자책기능

이책은오픈소스이며 GitHub 에호스팅되어있습니다: schoeberl/chisel-book. 이책에나온모든 Chisel 코드예제는저장소에포함되어있습니다. 책에나온모든코드는컴파일러를통과했으므로구문오류가없어야합니다. 더나아가대부분의예제는테스트벤치도포함하고있습니다. 코드는해당소스로부터자동으로추출됩니다. 더규모가큰 Chisel 예제들은함께제공되는저장소인 chisel-examples와 ip-contributions에모아두었습니다.

책에서오류나오타를발견하신경우, GitHub 풀리퀘스트가개선사항을반영하는가장편리한방법입니다. GitHub 에이슈를등록하거나예전방식대로평범한이메일을보내는방법으로도개선을위한피드백이나의견을제공하실수있습니다.

이책의저장소에는제가덴마크공과대학교에서 13 주과정인 Digital Electronics[5]에사용하는 Latex 슬라이드도들어있습니다. 그과정에는 Chisel 로작성된 실습과제도포함되어있습니다. Chisel 로디지털설계를가르치고계신다면슬라이드와실습과제를필요에맞게자유롭게변형하셔도됩니다. 모든자료는오픈소스입니다. 책과슬라이드를빌드하려면최신버전의 LaTeX 와 Chisel 에필요한도구 (sbt 와 Java JDK 설치) 가필요합니다. 모든코드는컴파일, 테스트, 추출되며 LaTeX 는다음의간단한명령으로컴파일됩니다:

```
$ make
```

[4] This function needs the Eclipse plugin for sbt.
[5] The course page contains the PDF versions of the slides

이책은 PDF 전자책으로무료로이용하실수있으며, Amazon에서전통적인인쇄판으로도구매하실수있습니다. 전자책버전은추가자료와 위키백과 항목에대한링크를제공합니다. 이책에직접담기에는적합하지않은배경정보 (예: 이진수체계) 에는위키백과항목을사용하고있습니다. 저희는전자책의형식을 iPad 와같은태블릿에서읽기에최적화했습니다.

1.6 더읽을거리

다음은디지털설계와 Chisel 에관한더읽을거리목록입니다:

- William J. Dally 와 R. Curtis Harting 가쓴 Digital Design: A Systems Approach는디지털설계에관한교과서입니다. 이책은하드웨어기술언어로 Verilog 를사용하는버전과 VHDL 을사용하는버전, 두가지로제공됩니다.

공식 Chisel 문서와추가문서들은온라인에서확인하실수있습니다:

- Chisel 홈페이지는 Chisel 을내려받고학습하는공식출발점입니다.
- 덴마크공과대학교의 Digital Electronics 2 과정웹사이트에는 Chisel 을기반으로한 13 주과정의슬라이드가실려있습니다. 슬라이드의소스코드는이책의소스코드의일부로제공됩니다. 필요에맞게자유롭게변형하여가르치는데사용하셔도됩니다.
- schoeberl/chisel-lab GitHub 저장소에는 Digital Electronics 2 과정을위한 Chisel 실습과제가들어있습니다. 이실습과제는이책으로독학하는데도잘맞습니다.
- 빈 Chisel 프로젝트는매우최소한의하드웨어 (가산기) 와테스트를갖춘좋은출발점입니다. 그프로젝트는여러분의 GitHub 저장소의기반으로삼을수있는 GitHub 템플릿입니다.
- Chisel3 치트시트는 Chisel 의주요구성요소들을한페이지로요약하고있습니다.
- Scott Beamer 의 Agile Hardware Design 과정에는고급 Chisel 예제가들어있습니다. 강의자료에는실행가능한소스예제가포함되어있으며 Jupyter 노트북형태로제공됩니다.
- ChiselTest는별도의저장소에있습니다.

- Generator Bootcamp는하드웨어생성기에초점을맞춘 Chisel 과정으로, Jupyter 노트북형태로제공됩니다
- Chisel Tutorial은테스터와풀이를포함한작은연습문제들로구성된바로사용가능한설정프로젝트를제공합니다. 다만다소오래된내용입니다.
- Christopher Celio 가작성한 Chisel 스타일가이드입니다.

1.7 연습문제

각장은실습형연습문제로마무리됩니다. 도입부연습문제에서는 FPGA 보드를사용하여 LED 하나를깜빡이게해보겠습니다.[6] 첫단계로, GitHub 에서 chisel-examples 저장소를클론 (또는포크) 하십시오. Hello World 예제는최소구성프로젝트로설정된 hello-world 폴더에있습니다. src/main/scala/Hello.scala 에서깜빡이는 LED 의 Chisel 코드를살펴볼수있습니다. 다음단계에따라깜빡이는 LED 를컴파일하십시오.

```
$ git clone https://github.com/schoeberl/chisel-examples.git
$ cd chisel-examples/hello-world/
$ sbt run
```

Chisel 컴포넌트를초기다운로드한후, 이과정은 Verilog 파일 Hello.v 를생성합니다. 이 Verilog 파일을살펴보십시오. 이파일에는두개의입력 clock 과 reset, 그리고하나의출력 io_led 가포함되어있음을확인할수있습니다. 이 Verilog 파일을 Chisel 모듈과비교해보면, Chisel 모듈에는 clock 이나 reset 이포함되어있지않다는것을알수있습니다. 이러한신호는암묵적으로생성되며, 대부분의설계에서는이러한저수준의세부사항을다룰필요가없다는것이편리합니다. Chisel 은레지스터컴포넌트를제공하며, 이들은필요한경우자동으로 clock 과 reset 에연결됩니다.

다음단계는합성도구용 FPGA 프로젝트파일을설정하고, 핀을할당하고, Verilog 코드를컴파일[7]한후, 그결과로생성된비트파일로 FPGA 를구성하는것입니다. 이러한단계의세부사항은이책에서제공할수없습니다. Intel Quartus 또는 AMD Vivado 도구의매뉴얼을참고하시기바랍니다. 다만, 예제저장소에

[6] 현재사용가능한 FPGA 보드가없다면, 연습문제마지막에시뮬레이션버전을보여드릴것이므로계속읽어나가시기바랍니다.

[7] 실제과정은로직합성, 배치및배선 (place and route) 수행, 타이밍분석수행, 비트파일생성이라는단계를거치며더정교합니다. 그러나이도입부예제의목적을위해서는코드를그저" 컴파일" 한다고부르겠습니다.

는여러인기있는 FPGA 보드 (예: DE2-115) 를위한즉시사용가능한 Quartus 프로젝트가 quartus 폴더에포함되어있습니다. 저장소에사용중인보드에대한지원이포함되어있다면, Quartus 를실행하고, 프로젝트를열고, *Play* 버튼을눌러컴파일한다음, *Programmer* 버튼으로 FPGA 보드를구성하면 LED 중하나가깜빡여야합니다.

축하합니다! Chisel 로만든첫번째설계를 FPGA 에서실행하는데성공하셨습니다!

LED 가깜빡이지않는다면리셋상태를확인하십시오. DE2-115 구성에서는리셋입력이 SW0 에연결되어있습니다.

이제깜빡임주파수를더느리거나더빠른값으로변경하고빌드과정을다시실행해보십시오. 깜빡임주파수와깜빡임패턴은서로다른" 감정" 을전달합니다. 예를들어, 느리게깜빡이는 LED 는모든것이정상임을알리고, 빠르게깜빡이는 LED 는경보상태를알립니다. 어떤주파수가이두가지서로다른감정을가장잘표현하는지탐구해보십시오.

연습문제에대한좀더도전적인확장으로, 다음과같은점멸패턴을생성해보십시오: LED 가매초마다 200 ms 동안켜져있어야합니다. 이패턴을위해서는 LED 점멸의변화를카운터리셋으로부터분리해야할수도있습니다. blkReg 레지스터의상태를변경하는두번째상수가필요할것입니다. 이패턴은어떤종류의감정을만들어냅니까? 경보처럼느껴집니까, 아니면생존신호에더가깝습니까?

아직 FPGA 보드가없더라도, 깜빡이는 LED 예제를실행해볼수있습니다. 이경우 Chisel 시뮬레이션을사용하게됩니다. 시뮬레이션시간이너무길어지지않도록, Chisel 코드의클록주파수를 50000000 에서 50000 으로변경하십시오. 깜빡이는 LED 를시뮬레이션하려면다음명령을실행하십시오.

```
$ sbt test
```

이명령은백만클록사이클동안실행되는테스터를실행합니다. 깜빡임주파수는시뮬레이션속도에좌우되며, 이는다시컴퓨터의속도에좌우됩니다. 따라서시뮬레이션된깜빡이는 LED 를확인하기위해가정한클록주파수를약간조정해가며실험해야할수도있습니다.

2 기본컴포넌트

이절에서는디지털설계를위한기본컴포넌트인조합회로와플립플롭을소개합니다. 이러한필수요소들은더크고흥미로운회로를만들기위해조합될수있습니다.

디지털시스템은일반적으로이진신호를사용하며, 이는하나의비트또는신호가두가지가능한값중하나만가질수있음을의미합니다. 이러한값은흔히 0 과 1 로불립니다. 그러나저 (low)/고 (high), 거짓 (false)/참 (true), 비활성화 (deasserted)/활성화 (asserted) 와같은용어도사용합니다. 이러한용어들은이진신호의동일한두가지가능한값을의미합니다.

2.1 Chisel 의타입과상수

Chisel 은연결, 조합논리, 레지스터를기술하기위해 Bits, UInt, SInt 라는세가지데이터타입을제공합니다. UInt 와 SInt 는 Bits 를확장하며, 세가지타입모두비트벡터를나타냅니다. UInt 는이비트벡터에부호없는정수라는의미를부여하고, SInt 는부호있는정수라는의미를부여합니다.[1] Chisel 은부호있는정수표현으로 2 의보수를사용합니다. 다음은서로다른타입에대한정의로, 8 비트 Bits, 8 비트부호없는정수, 10 비트부호있는정수입니다.

```
Bits(8.W)
UInt(8.W)
SInt(10.W)
```

비트벡터의폭은 Chisel 의폭타입 (Width) 으로정의됩니다. 다음표현식은 Scala 정수 n 을 Chisel width 로캐스팅하며, 이는 Bits 벡터를정의하는데사용됩니다.

```
n.W
Bits(n.W)
```

상수는 Scala 정수를사용하고이를 Chisel 타입으로변환함으로써정의할수있습니다.

[1] 현재버전의 Chisel 에서 Bits 타입은연산이부족하여사용자코드에그다지유용하지않습니다.

```
0.U  // defines a UInt constant of 0
-3.S // defines a SInt constant of -3
```

상수는 Chisel 의폭타입을사용하여폭을지정하여정의할수도있습니다.

```
3.U(4.W) // An 4-bit constant of 3
```

3.U 와 4.W 라는표기법이다소재미있게느껴진다면, 이를타입이있는정수상수의한변형으로생각하십시오. 예를들어, C, Java, Scala 에서롱정수상수를나타내는 3L 과같은것입니다.

발생가능한함정: 전용비트폭으로상수를정의할때발생할수있는오류중하나는폭지정자 .W 를누락하는것입니다. 예를들어, 1.U(32) 는 1 을나타내는 32 비트폭상수를정의하지 않습니다. 대신표현식 (32) 는 32 번위치에서의비트추출로해석되어, 결과적으로 0 인단일비트상수가됩니다. 이는아마도프로그래머의원래의도가아닐것입니다.

Chisel 은 Scala 의타입추론의혜택을받으며, 많은곳에서타입정보를생략할수있습니다. 이는비트폭에도마찬가지로적용됩니다. 많은경우, Chisel 은올바른폭을자동으로추론합니다. 따라서 Chisel 로작성한하드웨어기술은 VHDL 이나 Verilog 보다더간결하고가독성이좋습니다.

십진법이아닌다른진법으로정의된상수의경우, 16 진법 (16 진수) 에는접두사 h, 8 진법 (8 진수) 에는 o, 2 진법 (2 진수) 에는 b 를붙인문자열로상수를정의합니다. 다음예제는서로다른진법으로상수 255 를정의하는방법을보여줍니다. 이예제에서는비트폭을생략했으며, Chisel 이상수를담기에충분한최소폭, 이경우에는 8 비트를추론합니다.

```
"hff".U        // hexadecimal representation of 255
"o377".U       // octal representation of 255
"b1111_1111".U // binary representation of 255
```

위코드는상수를나타내는문자열에서밑줄을사용하여자릿수를묶는방법을보여줍니다. 밑줄은무시됩니다.

텍스트를나타내는문자 (ASCII 인코딩) 도 Chisel 에서상수로사용할수있습니다.

```
val aChar = 'A'.U
```

논리값을나타내기위해, Chisel 은 Bool 타입을정의합니다. Bool 은 참또는 거짓값을나타낼수있습니다. 다음코드는 Bool 타입의정의와, Scala 의 Boolean 상수 true 와 false 를 Chisel Bool 상수로변환하여 Bool 상수를정의하는방법을보여줍니다.

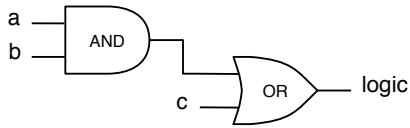


Figure 2.1: (a & b) | c 라는표현식에대한논리회로입니다. 배선은단일비트일수도있고여러비트일수도있습니다. Chisel 표현식과회로도는동일합니다.

```
Bool()
true.B
false.B
```

2.2 조합회로

Chisel 은 C, Java, Scala 및여러다른프로그래밍언어에서정의된방식과같은불대수 연산자를사용하여조합회로를기술합니다: & 는 AND 연산자이고, | 는 OR 연산자입니다. 다음코드줄은신호 a 와 b 를 *and* 게이트로결합하고, 그결과를신호 c 와 *or* 게이트로결합하여 logic 이라고이름붙인회로를정의합니다.

```
val logic = (a & b) | c
```

그림 2.1은이조합표현식의회로도를보여줍니다. 이회로는단일배선을 AND 및 OR 회로로결합한것뿐아니라, 비트벡터에대한것일수도있음에유의하십시오.

이예제에서는신호 logic 의타입도폭도정의하지않습니다. 둘다표현식의타입과폭으로부터추론됩니다. Chisel 의표준논리연산은다음과같습니다.

```
val a_and_b = a & b // bitwise and of a and b
val a_or_b = a | b  // bitwise or of a and b
val a_xor_b = a ^ b // bitwise xor of a and b
val a_not = ~a      // bitwise negation of a
```

산술연산은표준연산자를사용합니다.

```
val a_plus_b = a + b // addition of a and b
val a_minus_b = a - b // subtraction of b from a
val neg_a = -a      // negate a
val a_mul_b = a * b // multiplication of a and b
```

```
val a_div_b = a / b // division of a by b
val a_mod_b = a % b // modulo operation of a by b
```

연산결과의폭은, 덧셈과뺄셈의경우두피연산자중최대폭이고, 곱셈의경우두폭의합이며, 나눗셈과나머지연산의경우보통분자의폭입니다.[2]

신호는먼저어떤타입의 Wire 로정의될수도있습니다. 이후, := 갱신연산자로배선에값을대입할수있습니다.

```
val w = Wire(UInt())

w := a & b
```

단일비트는다음과같이추출할수있습니다:

```
val sign = x(31)
```

서브필드는끝위치에서시작위치까지추출할수있습니다:

```
val lowByte = largeWord(7, 0)
```

비트필드는 ## 연산자로연결됩니다.[3]

```
val word = highByte ## lowByte
```

표 2.1는전체연산자목록을보여줍니다 (내장연산자도참고하십시오). Chisel 의연산자우선순위는회로의평가순서에의해결정되며, 이는 Scala 연산자우선순위를따릅니다. 확신이서지않는다면항상괄호를사용하는것이좋은습관입니다.[4]

표 2.2는 Chisel 데이터타입에대해, 그리고 Chisel 데이터타입을위해정의된다양한함수를보여줍니다.

2.2.1 멀티플렉서

멀티플렉서는여러대안중하나를선택하는회로입니다. 가장기본적인형태에서는

[2] 정확한세부사항은FIRRTL 명세에서확인할수있습니다.

[3] 동일한연산을 Cat(highByte, lowByte) 로수행하는 Cat 함수도사용할수있음에유의하십시오.

[4] Chisel 에서의연산자우선순위는 Scala 연산자를실행하여하드웨어노드의트리를생성하는하드웨어엘라보레이션의부수효과입니다. Scala 의연산자우선순위는 Java/C 와유사하지만동일하지는않습니다. Verilog 는 C 와동일한연산자우선순위를가지지만, VHDL 은다른우선순위를가집니다. Verilog 는논리연산에대한우선순위순서를가지고있지만, VHDL 에서는이러한연산자들이동일한우선순위를가지며왼쪽에서오른쪽으로평가됩니다.

연산자	설명	데이터타입
* / %	곱셈, 나눗셈, 나머지	UInt, SInt
+ -	덧셈, 뺄셈	UInt, SInt
=== =/=	같음, 같지않음	UInt, SInt, Bool 을반환
> >= < <=	비교	UInt, SInt, Bool 을반환
« »	왼쪽시프트, 오른쪽시프트 (SInt 는부호확장)	UInt, SInt
~	NOT	UInt, SInt, Bool
& | ^	AND, OR, XOR	UInt, SInt, Bool
!	논리 NOT	Bool
&& ||	논리 AND, OR	Bool

Table 2.1: Chisel 에서정의한하드웨어연산자입니다.

함수	설명	데이터타입
v.andR v.orR v.xorR	AND, OR, XOR 축약	UInt, SInt, Bool 을반환
v(n)	단일비트추출	UInt, SInt
v(end, start)	비트필드추출	UInt, SInt
Fill(n, v)	비트스트링을 n 번복제	UInt, SInt
a ## b	비트필드연결	UInt, SInt
Cat(a, b, ...)	비트필드연결	UInt, SInt

Table 2.2: v 에대해호출되는, Chisel 에서정의한하드웨어함수입니다.

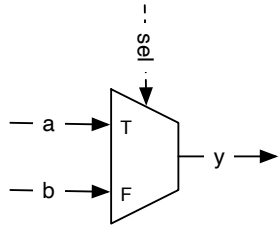


Figure 2.2: 기본적인 2:1 멀티플렉서입니다.

두개의대안중하나를선택합니다. 그림 2.2는이러한 2:1 멀티플렉서, 줄여서먹스 (mux) 를보여줍니다. 선택신호 (sel) 의값에따라신호 y 는신호 a 또는신호 b 를나타냅니다.

멀티플렉서는논리로구성할수있습니다. 하지만멀티플렉싱은매우표준적인연산이므로, Chisel 은멀티플렉서를제공합니다.

```
val result = Mux(sel, a, b)
```

여기서 sel 이 true.B 이면 a 가선택되고, 그렇지않으면 b 가선택됩니다. sel 의타입은 Chisel Bool 이며, 입력 a 와 b 는동일한타입인한어떤 Chisel 기본타입이나집합체 (번들또는벡터) 도될수있습니다.

논리연산과산술연산, 그리고멀티플렉서를이용하면모든조합회로를기술할수있습니다. 그러나 Chisel 은조합회로를더우아하게기술할수있도록추가적인컴포넌트와제어추상화를제공하며, 이는 5 장에서설명합니다.

디지털회로를기술하는데필요한두번째기본컴포넌트는레지스터라고도불리는상태요소이며, 다음에서설명합니다.

2.3 레지스터

Chisel 은 D 플립플롭의집합인레지스터를제공합니다. 레지스터는암묵적으로전역클록에연결되며상승에지에서갱신됩니다. 레지스터선언시초기화값이제공되면, 전역리셋신호에연결된동기리셋을사용합니다. 레지스터는비트의집합으로표현될수있는어떤 Chisel 타입도될수있습니다. 다음코드는리셋시 0 으로초기화되는 8 비트레지스터를정의합니다.

```
val reg = RegInit(0.U(8.W))
```

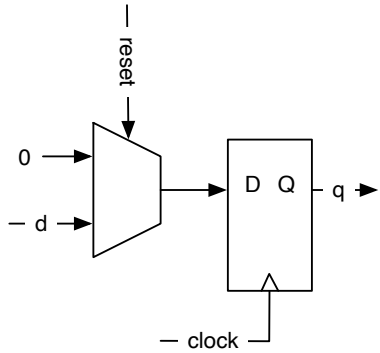


Figure 2.3: 0 으로의동기리셋을가진 D 플립플롭기반레지스터입니다.

입력은 := 갱신연산자로레지스터에연결되며, 레지스터의출력은표현식에서이름만으로사용할수있습니다.

```
reg := d
val q = reg
```

레지스터는정의시그입력에연결될수도있습니다.

```
val nextReg = RegNext(d)
```

그림 2.3은클록, 0.U 로의동기리셋, 입력 d, 출력 q 를가진우리의레지스터정의에대한회로를보여줍니다. 전역신호 clock 과 reset 은정의된각레지스터에암묵적으로연결됩니다.
레지스터는정의시그입력과초기값으로서의상수모두에연결될수도있습니다.

```
val bothReg = RegNext(d, 0.U)
```

조합논리와레지스터를나타내는신호를구별하기위해, 흔히레지스터이름뒤에 Reg 를붙이는관행이있습니다. Java 와 Scala 에서유래한또다른흔한관행은여러단어로구성된식별자에 CamelCase를사용하는것입니다. 관례상함수와변수는소문자로시작하고, Module 이름같은클래스 (타입) 는대문자로시작합니다.
Chisel 에서는식별자이름을비교적자유롭게지을수있습니다. 다만취향을살리고설명적인이름을사용하십시오. 또한여러단어가예약되어있으며, 이는부록 B에나열되어있습니다.

2.3.1 카운팅

카운팅은디지털시스템의기본적인연산입니다. 이벤트를세는데사용될수도있습니다. 하지만더흔하게는시간간격을정의하는데카운팅이사용됩니다. 클록사이클을세다가그시간간격이만료되면동작을트리거하는것입니다.

간단한접근법은어떤값까지세어올라가는것입니다. 하지만컴퓨터과학과디지털설계에서는 0 부터세기시작합니다. 따라서클록사이클 10 개를세고자한다면, 0 부터 9 까지셉니다. 다음코드는 9 까지센후 9 에도달하면다시 0 으로돌아가는카운터를보여줍니다.

```
val cntReg = RegInit(0.U(8.W))

cntReg := Mux(cntReg === 9.U, 0.U, cntReg + 1.U)
```

2.4 번들과 Vec 을사용한구조

Chisel 은관련된신호를그룹화하는두가지구성체를제공합니다: (1) Bundle 과 (2) Vec 입니다. Bundle 은서로다른타입의신호를이름이붙은필드로그룹화합니다. Vec 은같은타입의신호 (원소) 로이루어진인덱싱가능한컬렉션을나타냅니다. Bundle 과 Vec 은새로운사용자정의 Chisel 타입을생성하며임의로중첩될수있습니다.

2.4.1 번들

Chisel 의 Bundle 은여러신호를그룹화합니다. 번들전체를하나로참조할수도있고, 개별필드를이름으로접근할수도있습니다. Bundle 은 C 와 SystemVerilog 의 struct 나 VHDL 의 record 와유사합니다. 번들 (신호의모음) 은 Bundle 을확장하는클래스를정의하고, 생성자블록안에필드를 val 로나열함으로써정의할수있습니다.

```
class Channel() extends Bundle {
  val data = UInt(32.W)
  val valid = Bool()
}
```

번들을사용하려면 new 로생성한뒤 Wire 로감쌉니다. 필드는점표기법으로접근합니다.

```
val ch = Wire(new Channel())
ch.data := 123.U
ch.valid := true.B

val b = ch.valid
```

점표기법은객체지향언어에서흔히사용되며, x.y 는 x 가어떤객체에대한참조이고 y 가그객체의필드임을의미합니다. Chisel 은객체지향적이므로, 번들의필드에접근할때점표기법을사용합니다. 번들은전체로도참조할수있습니다.

```
val channel = ch
```

2.4.2 Vec

Chisel 의 Vec(벡터) 은같은타입의 Chisel 타입컬렉션을나타냅니다. 각요소는인덱스로접근할수있습니다. Chisel 의 Vec 은다른프로그래밍언어의배열데이터구조와유사합니다.[5]

Vec 은세가지다른목적으로사용됩니다. (1) 하드웨어에서의동적주소지정으로, 이는멀티플렉서입니다. (2) 레지스터파일로, 읽기의멀티플렉싱과쓰기의인에이블신호생성을포함합니다. (3) Module 의포트수를파라미터화하는것입니다. 하드웨어요소이든다른생성기데이터이든, 그밖의 것들의컬렉션에는 Scala 컬렉션인 Seq 를사용하는것이더좋습니다.

조합 Vec

Vec 은생성자에두개의매개변수, 즉요소개수와요소의타입을전달하여생성합니다. 조합 Vec 은 Wire 로감싸야합니다.

```
val v = Wire(Vec(3, UInt(4.W)))
```

개별요소는 (index) 로접근합니다. Wire 로감싼벡터는그저멀티플렉서일뿐입니다.

```
v(0) := 1.U
v(1) := 3.U
v(2) := 5.U
```

[5] Array 라는이름은이미 Scala 에서사용되고있습니다.

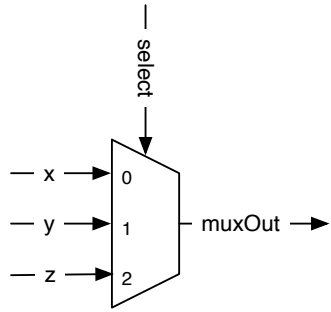


Figure 2.4: Wire 로감싼벡터는그저멀티플렉서일뿐입니다.

```
val index = 1.U(2.W)
val a = v(index)
```

다음은 Vec 을멀티플렉서로사용하는또다른예시입니다. 세개의입력은세개의와이어 x, y, z 에연결됩니다. select 와이어는어떤입력을사용할지선택하여 muxOut 에연결합니다.

```
val m = Wire(Vec(3, UInt(8.W)))
m(0) := x
m(1) := y
m(2) := z
val muxOut = m(select)
```

그림 2.4는위코드스니펫의결과로생성된회로도를보여줍니다.

WireDefault 를사용하는것과유사하게, VecInit 로 Vec 의기본값을설정할수있습니다. 다음코드는세개의상수기본값을갖는 3:1 멀티플렉서를나타냅니다. 첫번째상수로 UInt 데이터타입의크기 (3 비트) 를지정한다는점에유의하십시오. 조건 (cond) 을사용하면이러한기본값을덮어쓸수있습니다. 이덮어쓰기하드웨어자체는세개의 2:1 멀티플렉서로구성됩니다. 마지막줄은 defVec 멀티플렉서의세입력중하나를선택합니다. VecInit 는이미 Chisel 하드웨어를반환하므로 Wire 로감쌀필요가없다는점에유의하십시오.[6]

```
val defVec = VecInit(1.U(3.W), 2.U, 3.U)
when (cond) {
```

[6] 이는 Wire 로감싸야하는평범한 Vec 와는다릅니다. VecInit 를 WireDefault 로감쌀수도있지만, 이는흔치않은코딩스타일입니다.

```
    defVec(0) := 4.U
    defVec(1) := 5.U
    defVec(2) := 6.U
  }
  val vecOut = defVec(sel)
```

Vec 입력에 (WireDefault 에서처럼) 초기상수를설정하는것뿐만아니라, VecInit 로신호 (와이어) 를 Vec 의입력에연결할수도있습니다. 다음예제는 와이어 d, e, f 를 Vec 의세입력에연결합니다.

```
val defVecSig = VecInit(d, e, f)
val vecOutSig = defVecSig(sel)
```

레지스터 Vec

레지스터배열을정의하기위해 Vec 를레지스터로감쌀수도있습니다. 다음코드는 세개의레지스터로이루어진벡터를보여줍니다.

```
val vReg = Reg(Vec(3, UInt(8.W)))

val dout = vReg(rdIdx)
vReg(wrIdx) := din
```

그림 2.5는해당회로의회로도를보여줍니다. 이회로는세개의레지스터를포함합니다. 읽기인덱스 (rdIdx) 는세레지스터의출력에연결된멀티플렉서를선택합니다. 출력신호는 dout 입니다. 쓰기인덱스 (wrIdx) 는 din 으로부터의데이터로어느레지스터가기록될지를선택합니다. wrIdx 는레지스터들의세인에이블신호중하나를선택하는디코더를구동합니다.

다음예제는프로세서용레지스터파일을정의합니다. 각각 32 비트폭인 32 개의레지스터로, RISC-V의 32 비트버전과같은전형적인 32 비트 RISC 프로세서에서사용되는것과같습니다.

```
val registerFile = Reg(Vec(32, UInt(32.W)))
```

이레지스터파일의원소는인덱스로접근되며일반레지스터처럼사용됩니다.

```
registerFile(index) := dIn
val dOut = registerFile(index)
```

벡터의레지스터도초기화할수있습니다. 이는레지스터가리셋될때갖게되는

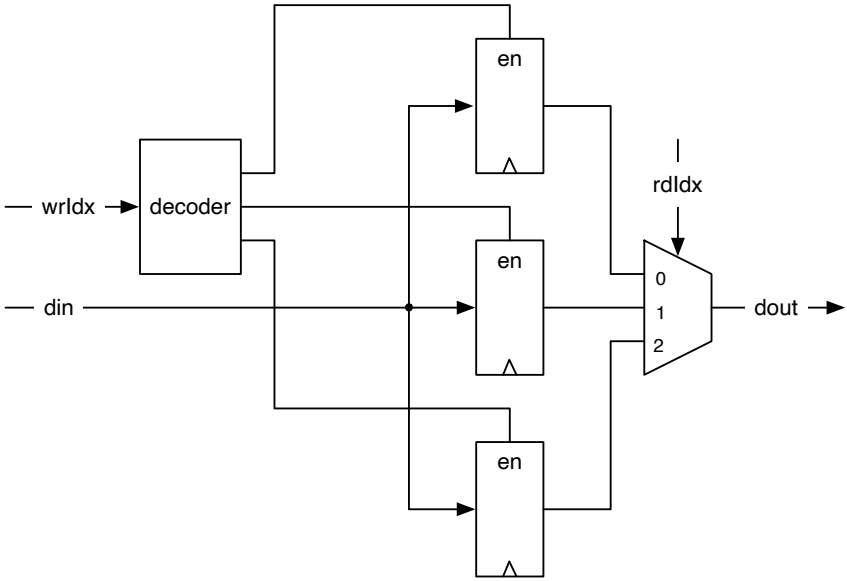


Figure 2.5: 레지스터로이루어진벡터입니다.

값이됩니다. 레지스터파일을초기화하기위해, 리셋용상수들을갖는 VecInit 를 RegInit 로감싸서사용합니다. 이세레지스터의입력은이후와이어 d, e, f 에연결됩니다.

```
val initReg = RegInit(VecInit(0.U(3.W), 1.U, 2.U))
val resetVal = initReg(sel)
initReg(0) := d
initReg(1) := e
initReg(2) := f
```

큰레지스터파일의모든원소를동일한값 (아마도 0) 으로리셋하고자한다면, Scala 시퀀스 Seq 를사용할수있습니다. VecInit 는 Chisel 타입을포함하는시퀀스로구성할수있습니다. Seq 는동일한값들로시퀀스를초기화하는생성함수 fill 을포함합니다. 다음코드는각각 32 비트폭이고 0 으로리셋되는 32 개의레지스터로이루어진레지스터파일을구성합니다.

```
val resetRegFile =
```

```
    RegInit(VecInit(Seq.fill(32)(0.U(32.W))))
val rdRegFile = resetRegFile(sel)
```

번들과벡터결합하기

번들과벡터는자유롭게섞어쓸수있습니다. 번들타입으로벡터를생성할때는벡터필드에대한프로토타입을전달해야합니다. 위에서정의한 Channel 을사용하면, 다음과같이채널로이루어진벡터를생성할수있습니다.

```
val vecBundle = Wire(Vec(8, new Channel()))
```

번들도마찬가지로벡터를포함할수있습니다.

```
class BundleVec extends Bundle {
  val field = UInt(8.W)
  val vector = Vec(4,UInt(8.W))
}
```

리셋값이필요한번들타입의레지스터를원할때는, 먼저해당번들의 Wire 를생성하고, 필요에따라개별필드를설정한다음, 이번들을 RegInit 에전달합니다:

```
val initVal = Wire(new Channel())

initVal.data := 0.U
initVal.valid := false.B

val channelReg = RegInit(initVal)
```

Bundle 과 Vec 의조합을사용하면우리만의데이터구조를정의할수있으며, 이는강력한추상화입니다.

발생가능한함정: Chisel 3 에서는부분할당이허용되지않지만, Chisel 2 에서는허용되었으며 Verilog 와 VHDL 에서도가능합니다. 다음코드는회로엘라보레이션중에오류를발생시킵니다:

```
val assignWord = Wire(UInt(16.W))

assignWord(7, 0) := lowByte
assignWord(15, 8) := highByte
```

이러한경우에는번들을사용하는것이더낫다는것이그논거입니다. 이문제에대한

한가지가능한해결책은 (지역) 번들을만들고, 그번들로부터 Wire 를생성하여개별필드에값을할당한다음, 해당번들을 asUInt 로 UInt 로캐스팅하고, 이값을목표 UInt 에할당하는것입니다. 여기서는지역적으로만필요하므로 Bundle 을지역데이터구조로정의하고있다는점에유의하십시오.

```
val assignWord = Wire(UInt(16.W))

class Split extends Bundle {
  val high = UInt(8.W)
  val low = UInt(8.W)
}

val split = Wire(new Split())
split.low := lowByte
split.high := highByte

assignWord := split.asUInt
```

이해결책의작은단점은번들필드가어떤순서로하나의비트벡터로병합되는지알아야한다는점입니다. 또다른방법은 Bool 의벡터를사용하여값을개별적으로할당한다음이를 UInt 로변환하는것입니다.

```
val vecResult = Wire(Vec(4, Bool()))

// example assignments
vecResult(0) := data(0)
vecResult(1) := data(1)
vecResult(2) := data(2)
vecResult(3) := data(3)

val uintResult = vecResult.asUInt
```

2.5 Wire, Reg, IO

UInt, SInt, Bits 는그자체로는하드웨어를나타내지않는 Chisel 타입입니다. 이들을 Wire, Reg, 또는 IO 로감쌀때만하드웨어가생성됩니다. Wire 는조합논리를나타내고, Reg 는레지스터 (D 플립플롭의집합) 를나타내며, IO 는 (구체적인 집적회로 (IC) 의핀처럼) 모듈연결을위한포트를나타냅니다. 어떤 Chisel 타

입이든 Wire, Reg, 또는 IO 로감쌀수있습니다.

하드웨어컴포넌트에이름을부여하려면이를 Scala 의불변변수에할당하면됩니다:[7]

```
val number = Wire(UInt())
val reg = Reg(SInt())
```

나중에 Chisel 연산자 := 를사용하여 Wire, Reg, 또는 IO 에값이나표현식을할당 (또는재할당) 할수있습니다

```
number := 10.U
reg := value - 3.U
```

Scala 의할당연산자 "=" 와 Chisel 연산자 ":=" 사이의작은차이에유의하십시오. 하드웨어객체를 생성할때 (그리고이름을부여할때) 는 Scala 의 "=" 연산자를사용하지만, 기존하드웨어객체에값을할당하거나재할당할때는 Chisel 의 ":=" 연산자를사용합니다.

조합값은조건부로할당될수있지만, 조건의모든분기에서할당되어야합니다. 그렇지않으면래치를기술하게되며, Chisel 컴파일러는이를거부합니다. 모범사례는 Wire 를생성할때기본값을정의하는것입니다. 따라서앞의코드는다음과같이다시작성하는것이더낫습니다.

```
val number = WireDefault(10.U(4.W))
```

Chisel 이신호와레지스터에필요한비트폭을추론하기는하지만, 하드웨어객체를생성할때의도한비트폭을명시하는것도좋은습관입니다. 대부분의경우리셋시 레지스터를알려진초기값으로설정하는것도좋은습관입니다:[8]

```
val reg = RegInit(0.S(8.W))
```

2.6 Chisel 은하드웨어를생성합니다

초기의 Chisel 코드를보면 Java 나 C 같은전통적인프로그래밍언어와비슷해보일수있습니다. 그러나 Chisel(또는다른어떤하드웨어기술언어든) 은실제로

[7] Scala 는 var 를사용한가변변수도지원하지만, Chisel 에서하드웨어를기술할때는이것이쓸모없습니다.

[8] 리셋시레지스터값을정의하지않은채로두면리셋배선의부하를다소줄일수있습니다. 그러나알려진리셋값을사용하면테스트와검증이단순해집니다.

하드웨어컴포넌트를정의합니다. 소프트웨어프로그램에서는코드의한줄한줄이차례로실행되는반면, 하드웨어에서는모든코드라인이 병렬로실행됩니다.

Chisel 코드가하드웨어를생성한다는점을명심하는것이필수적입니다. Chisel 회로기술에의해생성되는개별블록들을상상해보거나, 종이위에그려보십시오. 컴포넌트를생성할때마다하드웨어가추가되며, 할당문하나하나가게이트및/또는플립플롭을생성합니다.

좀더기술적으로말하면, Chisel 이여러분의코드를실행할때는 Scala 프로그램으로실행되며, Chisel 문들을실행함으로써하드웨어컴포넌트들을 수집하고그노드들을연결합니다. 이하드웨어노드들의네트워크가바로하드웨어이며, Chisel 은이를 ASIC 이나 FPGA 합성을위한 Verilog 코드로뽑아낼수도있고 Chisel 테스터로테스트할수도있습니다. 이하드웨어노드들의네트워크가완전히병렬로실행되는대상입니다.

소프트웨어엔지니어의입장에서보면, 애플리케이션을스레드로분할하고통신을위한잠금을올바르게처리할필요없이하드웨어에서만들어낼수있는이러한엄청난병렬성을상상해보십시오.

2.7 연습문제

서론에서여러분은 FPGA 보드에서 (chisel-examples에서가져온) 깜빡이는 LED 를구현했는데, 이는적절한하드웨어 *Hello World* 예제입니다. 이예제는내부상태와단일 LED 출력만사용했으며입력은사용하지않았습니다. 해당프로젝트를새폴더로복사한다음, `val sw = Input(UInt(2.W))` 로 `io Bundle` 에입력을몇개추가하여확장하십시오.

```
val io = IO(new Bundle {
  val sw = Input(UInt(2.W))
  val led = Output(UInt(1.W))
})
```

이스위치들을위해서는 FPGA 보드의핀도할당해야합니다. 핀할당의예시는 ALU 프로젝트의 Quartus 프로젝트파일에서찾아볼수있습니다 (예: DE2-115 FPGA 보드용).

이러한입력과핀할당을정의했다면, 간단한테스트부터시작하십시오: 설계에서모든깜박임로직을제거하고스위치하나를 LED 출력에연결한다음, 컴파일하고 FPGA 장치를구성하십시오. 스위치로 LED 를켜고끌수있습니까? 그렇다면이제입력을사용할수있게된것입니다. 그렇지않다면 FPGA 구성을디버깅해야합니다. 핀할당은도구의 GUI 버전으로도수행할수있습니다.

이제스위치두개를사용하여기본적인조합함수중하나를구현해보십시오. 예를 들어, 두스위치를 AND 연산하여그결과를 LED 에표시하십시오. 함수를바꿔 보십시오. 다음단계는멀티플렉서를구현하기위해입력스위치세개를사용하는것입니다: 하나는선택신호역할을하고, 나머지둘은 2:1 멀티플렉서의두입력이됩니다.

이제여러분은간단한조합함수를구현하고이를 FPGA 의실제하드웨어에서테스트할수있게되었습니다. 다음단계로, FPGA 구성을생성하기위한빌드프로세스가어떻게작동하는지처음으로살펴보겠습니다. 아울러 FPGA 를구성하거나 스위치를조작하지않고도회로를테스트할수있게해주는 Chisel 의간단한테스트 프레임워크도살펴보겠습니다.

3 빌드프로세스와테스트

더흥미로운 Chisel 코드를시작하려면먼저 Chisel 프로그램을컴파일하는방법, FPGA 에서실행하기위한 Verilog 코드를생성하는방법, 그리고디버깅용테스트를작성하고회로가올바른지검증하는방법을배워야합니다.

Chisel 은 Scala 로작성되어있으므로, Scala 를지원하는어떠한빌드프로세스든 Chisel 프로젝트에사용할수있습니다. Scala 용으로인기있는빌드도구중하나는 sbt로, Scala interactive build tool 의약자입니다. sbt 는빌드및테스트프로세스를진행할뿐만아니라, 올바른버전의 Scala 와 Chisel 라이브러리도내려받습니다.

3.1 sbt 로프로젝트빌드하기

Chisel 을나타내는 Scala 라이브러리와 Chisel 테스터는빌드프로세스중에 Maven 저장소에서자동으로내려받아집니다. 이라이브러리들은 build.sbt 에서참조됩니다. build.sbt 를 latest.release 로설정하여항상최신버전의 Chisel 을사용하도록구성할수도있습니다. 다만이렇게하면빌드할때마다 Maven 저장소에서버전을조회하게됩니다. 이조회에는빌드가성공하기위한인터넷연결이필요합니다. build.sbt 에서는 Chisel 과그밖의모든 Scala 라이브러리에대해특정버전을지정해사용하는편이더낫습니다. 때로는인터넷연결없이하드웨어코드를작성하고테스트할수있으면좋습니다. 예를들어, 비행기안에서하드웨어설계를하는것도멋진일입니다.

3.1.1 소스구성

sbt 는 Maven 빌드자동화도구로부터소스관례를물려받았습니다. Maven 은오픈소스 Java 라이브러리저장소도관리합니다.[1]

그림 3.1는일반적인 Chisel 프로젝트의소스트리구성을보여줍니다. 프로젝트의루트는프로젝트홈으로, build.sbt 를포함합니다. 빌드프로세스를위한

[1] 그곳이바로여러분이첫빌드에서 Chisel 라이브러리를내려받은곳이기도합니다: https://mvnrepository.com/artifact/edu.berkeley.cs/chisel3.

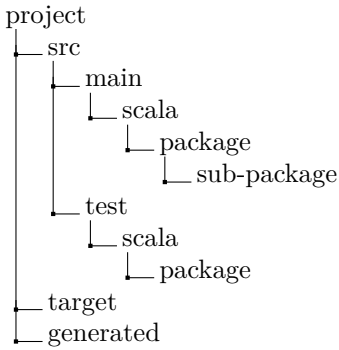


Figure 3.1: Chisel 프로젝트의소스트리구조 (sbt 사용)

Makefile, README, LICENSE 파일도포함될수있습니다. src 폴더는모든소스코드를포함합니다. 여기서부터하드웨어소스를포함하는 main 과테스터를포함하는 test 로나뉩니다. Chisel 이 Scala 를기반으로하므로두경우모두그다음폴더는 scala 입니다. 하드웨어생성기에유용할수있는 Java 코드를포함하고싶다면 java 폴더를추가하면됩니다. Chisel 은 Scala 로부터, Scala 는다시 Java 로부터 패키지 형태의소스구성을물려받습니다. 패키지는 Chisel 코드를네임스페이스로구성해줍니다. 패키지는하위패키지도포함할수있습니다. target 폴더는클래스파일과그밖의생성된파일을포함합니다. 생성된 Verilog 파일을위한폴더도사용하기를권장하는데, 보통 generated 라고부릅니다.

Chisel 에서네임스페이스기능을사용하려면클래스/모듈이패키지안에정의되어있다고선언해야합니다. 이예제에서는 mypack 패키지안입니다:

```
package mypack

import chisel3._

class Abc extends Module {
  val io = IO(new Bundle{})
}
```

이예제에서 Chisel 클래스를사용하기위해 chisel3 패키지를임포트하는것을볼수있다는점에유의하십시오.

다른컨텍스트 (패키지네임스페이스) 에서 Abc 모듈을사용하려면 mypack 패키지의컴포넌트들을임포트해야합니다. 밑줄 (_) 은와일드카드역할을하며, 이

는 mypack 의모든클래스가임포트됨을의미합니다.

```
import mypack._

class AbcUser extends Module {
  val io = IO(new Bundle{})

  val abc = new Abc()

}
```

mypack 의모든타입을임포트하지않고, 정규화된이름인 mypack.Abc 를사용하여 mypack 패키지안의 Abc 모듈을참조하는것도가능합니다.

```
class AbcUser2 extends Module {
  val io = IO(new Bundle{})

  val abc = new mypack.Abc()

}
```

클래스하나만임포트하여그인스턴스를생성하는것도가능합니다.

```
import mypack.Abc

class AbcUser3 extends Module {
  val io = IO(new Bundle{})

  val abc = new Abc()

}
```

3.1.2 sbt 실행하기

Chisel 프로젝트는간단한 sbt 명령으로컴파일하고실행할수있습니다.

```
$ sbt run
```

이명령은소스트리에있는모든 Chisel 코드를컴파일하고, main 메서드를가지거나 App 을확장한 object 를포함하는클래스를검색합니다. 그러한객체가둘이

상있으면모든객체가나열되며그중하나를선택할수있습니다. 실행할객체를 sbt 의매개변수로직접지정할수도있습니다.

```
$ sbt "runMain mypacket.MyObject"
```

기본적으로 sbt 는소스트리의 main 부분만검색하며 test 부분은검색하지않습니다.[2] ChiselTest 기반테스트를실행하려면다음과같이간단히실행하면됩니다.

```
$ sbt test
```

ChiselTest 관례를따르지않으면서 main 함수를포함하지만소스트리의 test 부분에위치한테스트가있다면, 다음과같은 sbt 명령으로실행할수있습니다.

```
$ sbt "test:runMain mypacket.MyMainTest"
```

3.1.3 Verilog 생성하기

Chisel 코드를 FPGA 나 ASIC 용으로합성하려면합성도구가이해할수있는하드웨어기술언어로 Chisel 을변환해야합니다. Chisel 을사용하면회로의합성가능한 Verilog 기술을생성할수있습니다.

Verilog 기술을생성하려면애플리케이션이필요합니다. App 을확장한 Scala 객체는애플리케이션이시작되는 main 함수를암묵적으로생성하는애플리케이션입니다. 이애플리케이션이하는유일한동작은새로운 Chisel 모듈—이예제에서는 Hello—을생성하여 Chisel 함수 emitVerilog() 에전달하는것입니다. 다음코드는 Verilog 파일 Hello.v 를생성합니다.

```
object Hello extends App {
  emitVerilog(new Hello())
}
```

emitVerilog() 의기본버전을사용하면생성된파일이프로젝트의루트폴더 (즉, sbt 명령을실행하는위치) 에놓이게됩니다. 생성된파일을하위폴더에넣으려면 emitVerilog() 에옵션을지정해야합니다. 그림 3.1에나온것처럼 generated 라는폴더를지정할것을권장합니다. 빌드옵션은두번째인수로지정할수있으며, 이는문자열배열입니다. 다음코드는하위폴더 generated 에 Verilog 파일 Hello.v 를생성합니다.

[2] 테스트폴더에는 main 을가진객체가아니라단위테스트가들어있는것이 Java/Scala 의관례입니다.

```
object HelloOption extends App {
  emitVerilog(new Hello(), Array("--target-dir",
      "generated"))
}
```

파일을작성하지않고도 Verilog 코드를 Scala String 으로요청할수있습니다. 테스트를위해그문자열을그냥출력해볼수있습니다.

```
object HelloString extends App {
  val s = getVerilogString(new Hello())
  println(s)
}
```

이러한형태의출력은웹기반 Scala 컴파일러이자런타임인 Scastie에서작은 Chisel 예제를보여줄때많이사용됩니다. 예제로는 Scastie 의 Hello World를참고하십시오.

3.1.4 도구흐름

그림 3.2는 Chisel 의도구흐름을보여줍니다. 디지털회로는 Hello.scala 로표시된 Chisel 클래스로기술됩니다. Scala 컴파일러는이클래스를 Chisel 및 Scala 라이브러리와함께컴파일하여, 표준 자바가상머신 (JVM)으로실행할수있는 Java 클래스파일 Hello.class 를생성합니다. 이클래스를 Chisel 드라이버로실행하면디지털회로의중간표현인 RTL 용유연한중간표현 (FIRRTL) 이생성됩니다. 이예제에서그파일은 Hello.fir 입니다. FIRRTL 컴파일러는회로에대한변환을수행합니다.

Treadle 은회로를시뮬레이션할수있는 FIRRTL 인터프리터입니다. Chisel 테스터와함께사용하면 Chisel 회로를디버깅하고테스트하는데쓸수있습니다. 어설션을사용하면테스트결과를제공할수있습니다. Treadle 은파형뷰어 (예: 무료뷰어인 GTKWave 나 Modelsim) 로볼수있는파형파일 (Hello.vcd) 을생성할수도있습니다.

FIRRTL 변환중하나인 Verilog 이미터는합성을위한 Verilog 코드 (Hello.v) 를생성합니다. 회로합성도구 (예: Intel Quartus, AMD/Xilinx Vivado, 또는 ASIC 도구) 가회로를합성합니다. FPGA 설계흐름에서는이도구가 FPGA 를구성하는데사용되는 FPGA 비트스트림, 예를들어 Hello.bit 를생성합니다.

그림 3.2는 Chisel 3.6 까지의 Chisel 도구흐름을보여준다는점에유의하십시오. 이후버전의 Chisel 은다른백엔드를사용합니다 (다음절참고).

이제 Chisel 프로젝트의기본구조와 sbt 로컴파일하고실행하는방법을알았으니, 간단한테스트프레임워크로넘어가겠습니다.

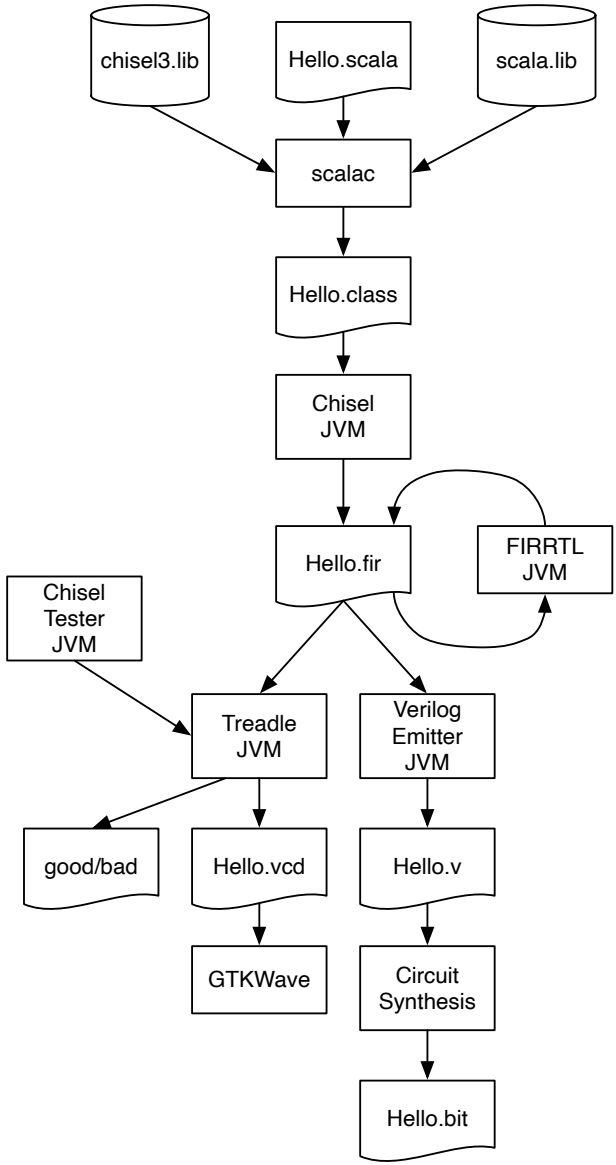


Figure 3.2: Chisel 생태계의도구흐름입니다.

3.1.5 Chisel 버전

Chisel 은 5.0.0 버전부터버전체계를 MAJOR.MINOR.PATCH 형식의시맨틱버저닝으로변경했습니다. Chisel 5.0.0 이전의체계는 3.MAJOR.MINOR 였습니다. Chisel 의마이너버전변경에는새로운기능이포함되지만, 하위호환방식으로이루어집니다. 메이저버전변경은호환성을깨는변경을도입할수있습니다. 이러한버전변경을강조하기위해 Chisel 은버전 4 를건너뛰었습니다.

Chisel 5 부터 Chisel 의백엔드는 FIRRTL 을위한 Scala 기반컴파일러에서 Verilog 를생성하기위해 firtool 을사용하는방식으로변경되었습니다. firtool 은더큰프로젝트인 LLVM CIRCT 프로젝트의일부입니다. "Chisel 3" 의최신버전은 3.6 이며 Scala 기반컴파일러와 firtool 두백엔드를모두포함합니다. 3.6 용 build.sbt 는다음과같습니다:

```
scalaVersion := "2.13.14"

scalacOptions ++= Seq( "-deprecation", "-feature",
    "-unchecked", "-language:reflectiveCalls", )

val chiselVersion = "3.6.1"
    addCompilerPlugin("edu.berkeley.cs" % "chisel3-plugin" %
    chiselVersion cross CrossVersion.full)
libraryDependencies += "edu.berkeley.cs" %% "chisel3" %
    chiselVersion
libraryDependencies += "edu.berkeley.cs" %% "chiseltest" %
    "0.6.1"
```

Chisel 버전 5 에서는 firtool 바이너리를수동으로설치해야합니다. Chisel 6 은 firtool 을 Chisel 라이브러리의일부로포함하므로권장되는최신버전의 Chisel 입니다.

CIRCT 로의변경은향후테스트인프라에영향을미칩니다. 향후테스트는 Verilator 위에서동작하는 Scala API 로수행될예정이며, 이는 Verilator 의설치를필수로만듭니다. 다만 CIRCT 프로젝트내에서 CIRCT 프레임워크안에서시뮬레이션하기위한개발이진행중입니다. 이도구는 Arcilator 라고불립니다. 그러나 ChiselTest 는 Chisel 5 와 Chisel 6 으로이식되었습니다. 새로운테스트프레임워크로의원활한전환을위해 Chisel 7 에호환성계층이제공될것으로예상됩니다.

Chisel 5 부터라이브러리는 org.chipsalliance 아래에게시됩니다. 따라서 build.sbt 는다음과같이변경되어야합니다:

```
scalaVersion := "2.13.14"
```

```
scalacOptions ++= Seq( "-deprecation", "-feature",
    "-unchecked", "-language:reflectiveCalls", )

val chiselVersion = "6.5.0"
    addCompilerPlugin("org.chipsalliance" % "chisel-plugin"
    % chiselVersion cross CrossVersion.full)
libraryDependencies += "org.chipsalliance" %% "chisel" %
    chiselVersion
libraryDependencies += "edu.berkeley.cs" %% "chiseltest" %
    "6.0.0"
```

이책은 Chisel 3.5, 3.6, 5, 6 을다루며, 해당버전들로테스트되었습니다.

Chisel, Scala, Java 버전

Chisel 은 Scala 라이브러리이므로 Scala 버전에의존합니다. Scala 자체는 Java 위에구축되어있으며 (Java 라이브러리를사용하고 JVM 에서실행됨) 따라서특정 Java 버전에의존합니다. Java 버전의변경은 Scala(패치) 버전의업데이트를필요로합니다. 안전한기준 Java 버전은 Java 8 이며, 이버전에서는 모든 Scala 및 Chisel 버전이동작합니다.[3] Chisel 자체는버전 5 까지 Scala 2.12 와 2.13 에서동작하도록유지되어왔습니다. Chisel 6 은 Scala 2.13 만을기반으로합니다.[4] 안전한선택은 Scala 2.13 을사용하는것입니다. 그러나 Chisel 은 Scala 컴파일러플러그인을사용하므로, 지원되는 Scala 의최고패치버전은게시된컴파일러플러그인에의해제한됩니다.[5] 표 3.1는 Chisel 버전과그에따른최고지원 Scala 및 Java 버전을보여줍니다.

3.1.6 GitHub 템플릿사용하기

chisel-empty 프로젝트는덧셈기회로, 테스터, 그리고 Verilog 코드를생성하고회로를테스트하고저장소를정리하기위한 Makefile 을포함하는최소한의 Chisel 프로젝트입니다. 이프로젝트는 GitHub 템플릿이며, 이는이템플릿을사용해새로운 GitHub 저장소를간단히시작할수있다는것을의미합니다.

[3] https://docs.scala-lang.org/overviews/jdk-compatibility/overview.html 참조
[4] Chisel 을 Scala 3 위에서동작하게하는작업은진행중입니다.
[5] https://mvnrepository.com/artifact/edu.berkeley.cs/chisel3-plugin 및 https://mvnrepository.com/artifact/org.chipsalliance/chisel-plugin 참조

Chisel	Scala	Java
3.5.6	2.13.10	17
3.6.1	2.13.14	22
5.3.x	2.13.14	22
6.5.x	2.13.14	22
6.6 ...		
6.7.x	2.13.16	24

Table 3.1: Chisel 과최고지원 Scala 및 Java 버전.

해당 GitHub 저장소로이동하여"Use this template" 버튼을눌러 GitHub 프로젝트를생성하십시오. 그런다음새프로젝트를로컬에클론하고 Makefile 을살펴보십시오. 다음명령으로 Verilog 코드를생성합니다:

```
make
```

다음명령으로덧셈기를테스트합니다:

```
make test
```

다음명령으로생성된모든파일을제거합니다:

```
make clean
```

sbt 와 git 명령을직접실행하여이러한작업을수행할수도있습니다.

3.2 Chisel 로테스트하기

하드웨어설계에대한테스트는일반적으로 테스트벤치라고부릅니다. 테스트벤치는테스트대상설계 (DUT) 를인스턴스화하고, 입력포트를구동하며, 출력포트를관찰하고, 이를예상값과비교합니다. Chisel 은 chiseltest 패키지에서 ChiselTest를제공합니다.

Chisel 의강점중하나는테스트벤치를작성할때 Scala 의모든기능을사용할 수있다는점입니다. 예를들어, 하드웨어의예상기능을소프트웨어시뮬레이터로 코딩하고하드웨어의시뮬레이션을소프트웨어시뮬레이션과비교할수있습니다. 이방법은프로세서구현을테스트할때매우효율적입니다 [19].

3.2.1 ScalaTest

ScalaTest는 Scala(및 Java) 용테스트도구입니다. ChiselTest 는 ScalaTest 의확장입니다. 따라서먼저간단한 ScalaTest 예제를살펴봅니다. 이를사용하려면다음줄을 build.sbt 에포함하여라이브러리를추가하십시오.

```
libraryDependencies += "org.scalatest" %% "scalatest" %
    "3.1.4" % "test"
```

테스트는보통 src/test/scala 에서찾을수있으며, 전체테스트스위트는다음과같이실행할수있습니다.

```
$ sbt test
```

Scala 정수덧셈과곱셈을테스트하는최소한의테스트 (테스트용 hello world) 는다음과같습니다.

```
import org.scalatest._
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should.Matchers

class ExampleTest extends AnyFlatSpec with Matchers {
  "Integers" should "add" in {
    val i = 2
    val j = 3
    i + j should be (5)
  }

  "Integers" should "multiply" in {
    val a = 3
    val b = 4
    a * b should be (12)
  }
}
```

ScalaTest 는실행가능한명세처럼읽히는간단한단위테스트를가능하게합니다. 위예제에는두개의테스트가포함되어있으며, 테스트실행결과는명세를반복해서보여주고두테스트가모두통과했음을나타냅니다.

```
[info] ExampleTest:
[info] Integers
[info] - should add
```

```
[info] Integers
[info] - should multiply
[info] ScalaTest
[info] Run completed in 119 milliseconds.
[info] Total number of tests run: 2
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 2, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[info] Passed: Total 2, Failed 0, Errors 0, Passed 2
```

sbt test 는사용가능한모든테스트를실행하며, 이는회귀테스트에유용합니다.[6] 그러나단일테스트 (스위트) 만실행하고자한다면다음과같이할수있습니다.

```
$ sbt "testOnly ExampleTest"
```

클래스이름의철자를잘못입력하면, 예를들어 ExampleTest 처럼입력하면, 비교적조용한오류메시지가나타납니다: No tests were executed.

3.2.2 ChiselTest

ChiselTest는 Scala 와 Java 용 ScalaTest 도구를기반으로한, Chisel 모듈을위한표준테스트도구이며, 이를사용하여 Chisel 테스트를실행할수있습니다. 이를사용하려면다음줄을 build.sbt 에포함하여 chiseltest 라이브러리를추가하십시오.

```
libraryDependencies += "edu.berkeley.cs" %% "chiseltest" %
    "0.5.6"
```

이방식으로 ChiselTest 를포함하면필요한버전의 ScalaTest 도자동으로포함됩니다. 따라서 ScalaTest 라이브러리를위한별도의줄을포함할필요가없습니다. ChiselTest 를사용하려면다음패키지들을임포트해야합니다.

```
import chisel3._
import chiseltest._
import org.scalatest.flatspec.AnyFlatSpec
```

회로를테스트하는데는 (적어도) 두가지구성요소가포함됩니다: 테스트대상장치 (흔히 DUT 라고부름) 와, 테스트벤치라고도불리는테스트로직입니다. 테스트는 sbt test 로시작됩니다. main 함수를가진객체는필요하지않습니다.

[6] 이책의저장소에서 sbt test 를실행해보면 90 개이상의테스트가통과하는것을볼수있습니다.

다음코드는우리의간단한테스트대상설계를보여줍니다. 이는두개의입력포트 (2 비트폭) 와두개의출력포트, 즉 2 비트폭포트하나와 Bool 포트하나를포함합니다. 이회로는입력 a 와 b 에대해비트단위 AND 연산을수행하여그결과를 out 에출력하고두신호의동등성을검사합니다.

```
class DeviceUnderTest extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(2.W))
    val b = Input(UInt(2.W))
    val out = Output(UInt(2.W))
    val equ = Output(Bool())
  })

  io.out := io.a & io.b
  io.equ := io.a === io.b
}
```

이 DUT 를위한테스트벤치는 ChiselScalatestTester 와함께 AnyFlatSpec 을확장하며, 이는 ScalaTest 내에서 ChiselTest 기능을제공합니다. test() 메서드는 DUT 를매개변수로, 테스트코드를함수리터럴로받아호출됩니다.

```
class SimpleTest extends AnyFlatSpec with
    ChiselScalatestTester {
  "DUT" should "pass" in {
    test(new DeviceUnderTest) { dut =>
      dut.io.a.poke(0.U)
      dut.io.b.poke(1.U)
      dut.clock.step()
      println("Result is: " + dut.io.out.peekInt())
      dut.io.a.poke(3.U)
      dut.io.b.poke(2.U)
      dut.clock.step()
      println("Result is: " + dut.io.out.peekInt())
    }
  }
}
```

DUT 의입력및출력포트는 dut.io 를통해접근합니다. 포트에 poke 를사용하여값을설정할수있으며, 이는입력포트의 Chisel 타입으로서의값을매개변수로받습니다. 출력포트는포트에대해 peekInt() 또는 peekBoolean() 을호출하여읽을수있으며, 이는 Scala 타입으로서의값을반환합니다. 테스터는 dut.clock.step()

으로시뮬레이션을한클록사이클진행시킵니다. 시뮬레이션을여러클록사이클진행시키려면 step() 에매개변수를제공할수있습니다. 출력값은 println() 으로출력할수있습니다.

테스트를실행하면

```
$ sbt "testOnly SimpleTest"
```

(다른정보와함께) 터미널에출력된결과를확인할수있습니다.

```
...
Result is: 0
Result is: 2
[info] SimpleTest:
[info] DUT
[info] - should pass
...
```

0 AND 1 의결과는 0 이고, 3 AND 2 의결과는 2 임을알수있습니다. 출력내용을직접확인하는것도좋은출발점이지만, 출력포트에대해 expect(value) 를호출하고예상값을매개변수로지정함으로써테스트벤치자체에기대값을명시할수도있습니다. 다음예제는기대값을사용한테스트를보여줍니다.

```
class SimpleTestExpect extends AnyFlatSpec with
    ChiselScalatestTester {
  "DUT" should "pass" in {
    test(new DeviceUnderTest) { dut =>
      dut.io.a.poke(0.U)
      dut.io.b.poke(1.U)
      dut.clock.step()
      dut.io.out.expect(0.U)
      dut.io.a.poke(3.U)
      dut.io.b.poke(2.U)
      dut.clock.step()
      dut.io.out.expect(2.U)
    }
  }
}
```

이테스트를실행해도하드웨어의값은전혀출력되지않지만, 모든기대값이올바르므로모든테스트가통과했음을알수있습니다.

```
[info] SimpleTestExpect:
```

```
[info] DUT
[info] - should pass
[info] ScalaTest
[info] Run completed in 1 second, 85 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[info] Passed: Total 1, Failed 0, Errors 0, Passed 1
```

DUT 나테스트벤치중하나에오류가있을경우테스트가실패하며, 기대값과실제값의차이를설명하는오류메시지가출력됩니다. 다음예에서는테스트벤치를 4 를기대하도록변경했는데, 이는오류입니다.

```
[info] SimpleTestExpect:
[info] DUT
[info] - should pass *** FAILED ***
[info]   io_out=2 (0x2) did not equal expected=4 (0x4)
              (lines in testing.scala: 27) (testing.scala:35)
[info] ScalaTest
[info] Run completed in 1 second, 214 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 0, failed 1, canceled 0, ignored 0, pending 0
[info] *** 1 TEST FAILED ***
[error] Failed: Total 1, Failed 1, Errors 0, Passed 0
[error] Failed tests:
[error]  SimpleTestExpect
```

peek() 함수는 Chisel 타입을반환하므로, Scala 타입으로사용하려면변환이필요합니다. Scala 쪽에서테스트값을더쉽게사용할수있도록, ChiselTest 는 peekInt() 와 peekBoolean() 을지원합니다. 다음테스트예제는 peekInt() 로출력을읽는데, 이함수는 Scala 정수[7]를반환하며이값은 assert() 문에서사용됩니다. 마찬가지로 equ 출력을 Scala Boolean 으로읽어 assert 문에서직접사용할수있습니다.

```
class SimpleTestPeek extends AnyFlatSpec with
    ChiselScalatestTester {
  "DUT" should "pass" in {
```

[7] 임의로넓은정수값을지원하기위해, 반환값은 Scala Int 가아니라 Scala BigInt 입니다.

```
    test(new DeviceUnderTest) { dut =>
      dut.io.a.poke(0.U)
      dut.io.b.poke(1.U)
      dut.clock.step()
      dut.io.out.expect(0.U)
      val res = dut.io.out.peekInt()
      assert(res == 0)
      val equ = dut.io.equ.peekBoolean()
      assert(!equ)
    }
  }
}
```

이예제는 DUT 의값을 Scala 타입으로읽어오는이점을확인하기에는다소단순합니다. 하지만어떤값이참이될때까지반복하는것과같이더복잡한테스트에서는이러한함수가유용해집니다.

이절에서는간단한테스트를위한 Chisel 의기본테스트기능을설명했습니다. 하지만테스터를작성할때 Scala 의모든기능을활용할수있다는점을기억하십시오. 예를들어, DUT 를검증하기위한참조모델을 Scala 로작성하는것도여기에포함됩니다.

3.2.3 파형

위에서설명한테스터는소규모설계와소프트웨어개발에서흔히볼수있는 단위테스트에잘맞습니다. 단위테스트의모음은 회귀테스트 역할도할수있습니다. 하지만더복잡한설계를디버깅하려면여러신호를한꺼번에살펴보고싶을것입니다. 디지털설계를디버깅하는전통적인방법은신호를파형으로표시하는것입니다. 파형에서는신호가시간에따라표시됩니다.

Chisel 테스터는모든레지스터와모든 IO 신호를포함하는파형을생성할수있습니다. 다음예제에서는앞선예제의 DeviceUnderTest(2 비트 AND 함수) 에대한파형테스터를보여줍니다. 테스트를위한파형을생성하려면, 다음 sbt 명령어에서보듯이테스트에 writeVcd=1 정의를전달하십시오.

```
sbt "testOnly SimpleTest -- -DwriteVcd=1"
```

무료뷰어인 GTKWave나 ModelSim 으로파형을볼수있습니다. GTKWave 를시작하고 *File – Open New Window* 를선택한다음, Chisel 테스터가 .vcd 파일을생성한폴더로이동하십시오. 기본적으로생성된파일은 test_run_dir 내부에, 테스트설명이름으로된하위폴더에위치합니다. 이폴더안

에서 DeviceUnderTest.vcd 를찾을수있을것입니다. 왼쪽에서신호를선택해메인창으로드래그할수있습니다. 신호구성을저장하려면 *File – Write Save File* 을사용하고, 나중에 *File – Read Save File* 로불러올수있습니다.

파형생성은 WriteVcdAnnotation 애너테이션을 test() 함수에전달하여시작할수도있습니다.[8] 먼저입력에값을 poke 하고 step 으로클록을진행시키는간단한테스터로시작하겠습니다. 여기서는출력을읽거나 expect 로비교하지않습니다. 대신생성된 .vcd 파일에서파형을살펴볼것입니다.

```
class WaveformTest extends AnyFlatSpec with
    ChiselScalatestTester {
  "Waveform" should "pass" in {
    test(new DeviceUnderTest)
      .withAnnotations(Seq(WriteVcdAnnotation)) { dut =>
      dut.io.a.poke(0.U)
      dut.io.b.poke(0.U)
      dut.clock.step()
      dut.io.a.poke(1.U)
      dut.io.b.poke(0.U)
      dut.clock.step()
      dut.io.a.poke(0.U)
      dut.io.b.poke(1.U)
      dut.clock.step()
      dut.io.a.poke(1.U)
      dut.io.b.poke(1.U)
      dut.clock.step()
    }
  }
}
```

가능한모든입력값을일일이나열하는것은확장성이없습니다. 따라서 DUT 를구동하기위해 Scala 코드를일부사용하겠습니다. 다음테스터는두개의 2 비트입력신호에대해가능한모든값을나열합니다.

```
class WaveformCounterTest extends AnyFlatSpec with
    ChiselScalatestTester {
  "WaveformCounter" should "pass" in {
    test(new DeviceUnderTest)
      .withAnnotations(Seq(WriteVcdAnnotation)) { dut =>
```

[8] 이는명령줄옵션을사용하는것의대안입니다.

```
      for (a <- 0 until 4) {
        for (b <- 0 until 4) {
          dut.io.a.poke(a.U)
          dut.io.b.poke(b.U)
          dut.clock.step()
        }
      }
    }
  }
}
```

그리고다음과같이실행합니다.

```
$ sbt "testOnly WaveformCounterTest"
```

3.2.4 printf 디버깅

또다른형태의디버깅은"printf 디버깅" 입니다. 이디버깅방법의이름은, 프로그램실행중관심있는변수를출력하기위해 C 코드에단순히 printf 문을넣는데서유래했습니다. 이 printf 디버깅은 Chisel 회로테스트중에도사용할수있습니다. 출력은클록의상승에지에서발생합니다. printf 문은 DUT 의 printf 디버깅버전에서보듯이모듈정의어디에든삽입할수있습니다.

```
class DeviceUnderTestPrintf extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(2.W))
    val b = Input(UInt(2.W))
    val out = Output(UInt(2.W))
  })

  io.out := io.a & io.b
  printf("dut: %d %d %d\n", io.a, io.b, io.out)
}
```

가능한모든값을순회하는카운터기반테스터로이모듈을테스트하면, 다음과같은출력을얻어 AND 함수가올바르다는것을확인할수있습니다.

```
Elaborating design...
Done elaborating.
dut: 0 0 0
dut: 0 1 0
```

```
dut: 0 2 0
dut: 0 3 0
dut: 1 0 0
dut: 1 1 1
dut: 1 2 0
dut: 1 3 1
dut: 2 0 0
dut: 2 1 0
dut: 2 2 2
dut: 2 3 2
dut: 3 0 0
dut: 3 1 1
dut: 3 2 2
dut: 3 3 3
dut: 0 0 0
test DeviceUnderTestPrintf Success: 0 tests passed in 18 cycles
in 0,031521 seconds 571,04 Hz
```

Chisel 의 printf 는 C 및 Scala 스타일포맷팅을지원합니다.

3.3 연습문제

이연습문제에서는 chisel-examples의깜빡이는 LED 를다시살펴보고 Chisel 테스트를탐구해보겠습니다.

3.3.1 최소프로젝트

먼저, 최소한의 Chisel 프로젝트가무엇인지알아봅시다. Hello World 예제의 파일들을탐구해보십시오. Hello.scala 는유일한하드웨어소스파일입니다. 이파일은깜빡이는 LED 의하드웨어기술 (class Hello) 과 Verilog 코드를생성하는 App 을포함합니다.

각파일은 Chisel 및관련패키지의 import 로시작합니다:

```
import chisel3._
```

그다음으로목록 1.1에나온것과같이하드웨어기술이이어집니다. Verilog 기술을생성하려면애플리케이션이필요합니다. 이애플리케이션이하는유일한동작은새 Hello 객체를생성하여 emitVerilog() 함수에전달하는것입니다.

```
object Hello extends App {
  emitVerilog(new Hello())
}
```

다음명령으로예제생성을수동으로실행하십시오

```
$ sbt run
```

그리고생성된 Hello.v 를편집기로살펴보십시오. 생성된 Verilog 코드는그다지읽기쉽지않을수있지만, 몇가지세부사항은파악할수있습니다. 파일은모듈 Hello 로시작하는데, 이는우리의 Chisel 모듈과같은이름입니다. 우리의 LED 포트는 output io_led 로식별할수있습니다. 핀이름은앞에 io_ 가붙은 Chisel 이름입니다. LED 핀외에도, 모듈에는 clock 과 reset 입력신호도포함되어있습니다. 이두신호는 Chisel 에의해자동으로추가됩니다.

나아가, 두레지스터 cntReg 와 blkReg 의정의도확인할수있습니다. 모듈정의끝부분에서이레지스터들의리셋및갱신도찾을수있습니다. Chisel 은동기식리셋을생성한다는점에유의하십시오.

sbt 가올바른 Scala 컴파일러와 Chisel 라이브러리를가져올수있으려면 build.sbt 가필요합니다:

```
scalaVersion := "2.13.10"

scalacOptions ++= Seq( "-feature",
    "-language:reflectiveCalls", )

val chiselVersion = "3.5.6"
    addCompilerPlugin("edu.berkeley.cs" %% "chisel3-plugin"
    % chiselVersion cross CrossVersion.full)
libraryDependencies += "edu.berkeley.cs" %% "chisel3" %
    chiselVersion
libraryDependencies += "edu.berkeley.cs" %% "chiseltest" %
    "0.5.6"
```

이예제에서는실행할때마다새버전을확인하는것을피하기위해구체적인 Chisel 버전번호를사용하고있다는점에유의하십시오 (인터넷에연결되어있지않을때, 예컨대비행중에하드웨어설계를할때는이확인이실패합니다). 또한 Chisel 3.5 부터필요한 Chisel3 컴파일러플러그인도추가했습니다. 라이브러리의존성을다음과같이변경하여최신 Chisel 버전을사용하도록 build.sbt 설정을바꿔보십시오

```
libraryDependencies += "edu.berkeley.cs" %% "chisel3" %
```

```
"latest.release"
```

그리고 sbt 로빌드를다시실행하십시오. 더새로운버전의 Chisel 이존재하며, 그것이자동으로다운로드됩니까?

편의를위해이프로젝트에는 Makefile 도들어있습니다. 이파일에는 sbt 명령만들어있어서그것을기억할필요가없으며, 다음명령으로 Verilog 코드를생성할수있습니다:

```
make
```

README 파일외에도, 이예제프로젝트에는다양한 FPGA 보드용프로젝트파일들도포함되어있습니다. 예를들어 quartus/altde2-115에는 DE2-115 보드용 Quartus 프로젝트를정의하는두개의프로젝트파일이있습니다. 주요정의들 (소스파일, 장치, 핀할당) 은일반텍스트파일 hello.qsf에서확인할수있습니다. 이파일을탐구하여어떤핀이어떤신호에연결되어있는지알아보십시오. 프로젝트를다른보드에맞게조정해야한다면, 바로이곳에서변경사항을적용합니다. Quartus 가설치되어있다면, 해당프로젝트를열고녹색 *Play* 버튼으로컴파일한다음 FPGA 를구성하십시오.

Hello World 는최소한의 Chisel 프로젝트라는점에유의하십시오. 더실제적인프로젝트들은소스파일이패키지로구성되어있으며테스터를포함합니다.

3.3.2 테스트연습문제

지난장의연습문제에서여러분은깜빡이는 LED 예제에입력을추가하여 AND 게이트와멀티플렉서를만들고이하드웨어를 FPGA 에서실행해보았습니다. 이제이예제를사용하여 Chisel 테스터로기능을테스트해서테스트를자동화하고 FPGA 보드에의존하지않도록해보겠습니다. 이전장의설계를사용하여 Chisel 테스터를추가하고기능을테스트하십시오. 가능한모든입력을열거하여 expect() 로출력을테스트해보십시오.

Chisel 내에서테스트를수행하면설계의디버깅속도를높일수있습니다. 그러나항상설계를 FPGA 용으로합성하여 FPGA 로테스트를실행해보는것이좋습니다. 그렇게하면설계의크기 (보통 LUT 와플립플롭단위로표현됨) 와최대클록주파수에서의설계성능에대해현실적인점검을수행할수있습니다. 참고로, 교과서적인스타일의파이프라인 RISC 프로세서는약 3000 개의 4 비트 LUT 를소비하며, 저가형 FPGA(Intel Cyclone 또는 Xilinx Spartan) 에서약 100 MHz 로동작할수있습니다.

4 컴포넌트

대규모디지털설계는흔히계층적인방식으로여러컴포넌트의집합으로구조화됩니다. 각컴포넌트는입력및출력와이어 (때로는포트라고도불림) 로이루어진인터페이스를가집니다. 이는집적회로 (IC) 의입출력핀과유사합니다. 컴포넌트는입력과출력을배선하여서로연결됩니다. 컴포넌트는계층을구성하기위해하위컴포넌트를포함할수있습니다. 칩의물리적핀에연결되는가장바깥쪽컴포넌트는최상위컴포넌트라고불립니다.

이장에서는 Chisel 에서컴포넌트가어떻게기술되는지설명하고, 컴포넌트의여러간단한예제를제공합니다. 이절의컴포넌트는컴포넌트를정의하고, 인스턴스화하고, 연결하는방법이라는원리를보여주기위한것일뿐이므로매우작다는 (예를들어그저가산기하나뿐인) 점에유의하십시오. 실제사례에서는가산기한줄보다훨씬더많은" 알맹이" 를포함하게됩니다.

4.1 Chisel 의컴포넌트는모듈입니다

하드웨어컴포넌트는 Chisel 에서모듈이라고불립니다. 각모듈은클래스 Module 을확장하며인터페이스를위한필드 io 를포함합니다. 인터페이스는 IO() 호출로감싸진 Bundle 로정의됩니다. Bundle 은모듈의입력포트와출력포트를나타내는필드들을포함합니다. 방향은필드를 Input() 또는 Output() 으로감싸서지정합니다. 방향은컴포넌트자체의관점을기준으로합니다.

여기서는두개의컴포넌트인가산기와레지스터로카운터를구성하는예제설계를보여드립니다. 그림 4.1는가산기컴포넌트의개략도를보여줍니다. 이컴포넌트는두개의입력 (a 와 b) 과하나의출력 (y) 을가집니다. 목록 4.1는가산기의 Chisel 정의를보여줍니다. 입력신호와출력신호는 io Bundle 의일부이므로 io.a 와같이 점표기법으로접근합니다.

그림 4.2는또다른간단한컴포넌트인 8 비트레지스터를보여줍니다. 이컴포넌트의 Chisel 코드는목록 4.2에나와있습니다.

이제이두컴포넌트로 0 부터 9 까지세고반복하는카운터를구성해보겠습니다. 그림 4.3는카운터의도식을보여줍니다. 가산기를사용하여 count 의값에 1 을더합니다. 멀티플렉서는이합과 0 중하나를선택합니다. next 라고불리는이결과

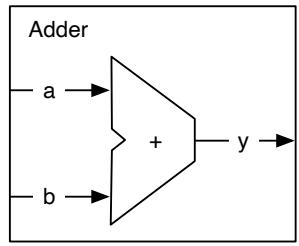


Figure 4.1: 가산기컴포넌트.

```
class Adder extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(8.W))
    val b = Input(UInt(8.W))
    val y = Output(UInt(8.W))
  })

  io.y := io.a + io.b
}
```

Listing 4.1: Chisel 의가산기컴포넌트.

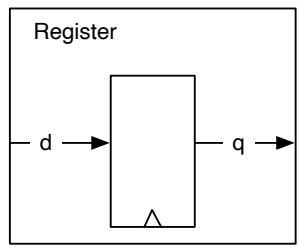


Figure 4.2: 레지스터컴포넌트.

```
class Register extends Module {
  val io = IO(new Bundle {
    val d = Input(UInt(8.W))
    val q = Output(UInt(8.W))
  })

  val reg = RegInit(0.U)
  reg := io.d
  io.q := reg
}
```

Listing 4.2: Chisel 의레지스터컴포넌트.

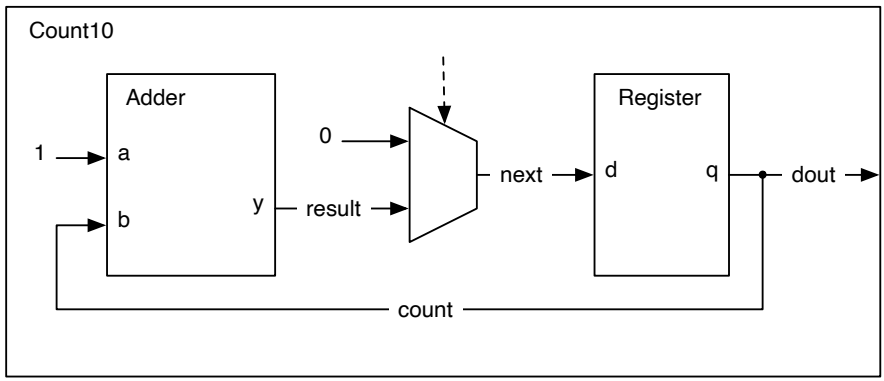


Figure 4.3: 컴포넌트로구성된카운터.

```
class Count10 extends Module {
  val io = IO(new Bundle {
    val dout = Output(UInt(8.W))
  })

  val add = Module(new Adder())
  val reg = Module(new Register())

  // the register output
  val count = reg.io.q
  // connect the adder
  add.io.a := 1.U
  add.io.b := count
  val result = add.io.y
  // connect the Mux and the register input
  val next = Mux(count === 9.U, 0.U, result)
  reg.io.d := next
  io.dout := count
}
```

Listing 4.3: 컴포넌트로구성된카운터.

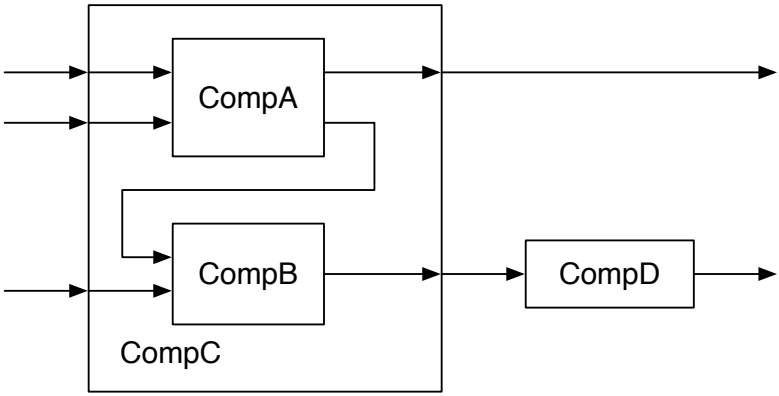


Figure 4.4: 컴포넌트계층으로구성된설계.

는레지스터컴포넌트의입력이됩니다. 레지스터의출력은 count 값이며, 동시에 Count10 컴포넌트의출력 (dout) 이기도합니다.

목록 4.3는 Count10 컴포넌트의 Chisel 코드를보여줍니다. 두컴포넌트는 new 로생성하고이를 Module() 로감싼뒤, add 와 reg 라는이름을부여함으로써인스턴스화됩니다. 이예제에서는레지스터의출력 (reg.io.q) 에 count 라는이름을부여합니다.

1.U 와 count 를가산기컴포넌트의두입력에연결합니다. 가산기컴포넌트의출력에는 result 라는이름을부여합니다. 멀티플렉서는현재카운터값 count 에따라 0.U 와 result 중하나를선택합니다. 멀티플렉서의출력에는 next 라는이름을부여하고이를레지스터컴포넌트의입력에연결합니다. 마지막으로, 카운터값 count 를 Count10 컴포넌트의유일한출력인 io.dout 에연결합니다.

4.2 중첩된컴포넌트

중간에서높은복잡도의하드웨어설계는중첩된컴포넌트들의계층으로구성됩니다. 그림 4.4는그러한예제설계의구조를보여줍니다. CompC 컴포넌트는세개의입력포트와두개의출력포트를가집니다. 이컴포넌트자체는 CompA 와 CompB 라는두개의하위컴포넌트로조립되며, 이들은 CompC 의입력및출력에연결됩니다. CompA 의한출력은 CompB 의입력에연결됩니다. CompD 컴포넌트는 CompC 컴포넌트와동일한계층수준에있으며이에연결됩니다.

```
class CompA extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(8.W))
    val b = Input(UInt(8.W))
    val x = Output(UInt(8.W))
    val y = Output(UInt(8.W))
  })

  // function of A
}

class CompB extends Module {
  val io = IO(new Bundle {
    val in1 = Input(UInt(8.W))
    val in2 = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })

  // function of B
}
```

Listing 4.4: 컴포넌트 CompA 와 CompB 의정의

```
class CompC extends Module {
  val io = IO(new Bundle {
    val inA = Input(UInt(8.W))
    val inB = Input(UInt(8.W))
    val inC = Input(UInt(8.W))
    val outX = Output(UInt(8.W))
    val outY = Output(UInt(8.W))
  })

  // create components A and B
  val compA = Module(new CompA())
  val compB = Module(new CompB())

  // connect A
  compA.io.a := io.inA
  compA.io.b := io.inB
  io.outX := compA.io.x
  // connect B
  compB.io.in1 := compA.io.y
  compB.io.in2 := io.inC
  io.outY := compB.io.out
}
```

Listing 4.5: 컴포넌트 CompC

목록 4.4는그림 4.4에나온두예제컴포넌트 CompA 와 CompB 의정의를보여줍니다. CompA 컴포넌트는 a 와 b 라는이름의두입력과 x 와 y 라는이름의두출력을가집니다. CompB 컴포넌트의포트에는 in1, in2, out 이라는이름을선택했습니다. 모든포트는비트폭 8 의부호없는정수 (UInt) 를사용합니다. 이예제코드는컴포넌트를연결하고계층을구성하는것에관한것이므로, 컴포넌트내부의구현은보여주지않습니다. 컴포넌트의구현은주석에"X 의기능" 이라고적힌위치에작성됩니다. 이예제컴포넌트들에는결부된기능이없으므로일반적인포트이름을사용했습니다. 실제설계에서는 data, valid, ready 와같이설명적인포트이름을사용하십시오.

목록 4.5에나온 CompC 컴포넌트는세개의입력포트와두개의출력포트를가집니다. 이컴포넌트는 CompA 와 CompB 컴포넌트로구성됩니다. CompA 와 CompB 가 CompC 의포트에어떻게연결되는지, 그리고 CompA 의출력포트와 CompB 의입력포트사이의연결도함께보여드립니다.

```
class CompD extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })

  // function of D
}
```

Listing 4.6: 컴포넌트 CompD

컴포넌트는 new 로, 예를들어 new CompA() 와같이생성되며, Module() 로감싸야합니다. 그모듈에대한참조는지역변수에저장되며, 이예제에서는 val compA = Module(new CompA()) 입니다.

이참조를통해모듈의 io 필드를역참조한다음, IO Bundle 의개별필드에접근하여 IO 포트에접근할수있습니다.

목록 4.6에나온, 우리설계에서가장단순한컴포넌트 (CompD) 는 in 이라는이름의입력포트하나와 out 이라는이름의출력포트하나만가지고있습니다. 예제설계에서마지막으로빠진부분은최상위컴포넌트인데, 이는그자체가컴포넌트 CompC 와 CompD 로조립되며, 목록 4.7에나와있습니다.

좋은컴포넌트설계는소프트웨어설계에서함수나메서드를잘설계하는것과비슷합니다. 주된질문중하나는컴포넌트에얼마나많은기능을넣어야하는지, 그리고컴포넌트가얼마나커야하는지입니다. 두극단은가산기와같은아주작은컴포넌트와, 전체마이크로프로세서와같은거대한컴포넌트입니다.

하드웨어설계초보자는흔히아주작은컴포넌트로시작합니다. 문제는디지털설계서적들이원리를보여주기위해아주작은컴포넌트를사용한다는점입니다. (그런책들에서도, 그리고이책에서도) 예제의크기는한페이지에들어맞고산만한세부사항을피하기위해작게유지됩니다.

컴포넌트의인터페이스는 (타입, 이름, 방향, IO 구성등으로인해) 다소장황합니다. 경험칙으로, 저는컴포넌트의핵심인함수가최소한컴포넌트의인터페이스만큼은길어야한다고제안합니다.

카운터와같은아주작은컴포넌트의경우, Chisel 은이를하드웨어를반환하는함수로기술하는더가벼운방법을제공합니다.

```
class TopLevel extends Module {
  val io = IO(new Bundle {
    val inA = Input(UInt(8.W))
    val inB = Input(UInt(8.W))
    val inC = Input(UInt(8.W))
    val outM = Output(UInt(8.W))
    val outN = Output(UInt(8.W))
  })

  // create C and D
  val c = Module(new CompC())
  val d = Module(new CompD())

  // connect C
  c.io.inA := io.inA
  c.io.inB := io.inB
  c.io.inC := io.inC
  io.outM := c.io.outX
  // connect D
  d.io.in := c.io.outY
  io.outN := d.io.out
}
```

Listing 4.7: 최상위컴포넌트

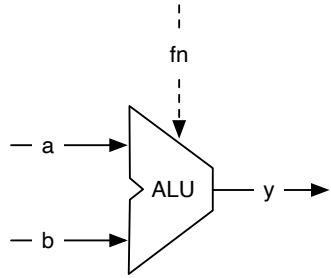


Figure 4.5: 산술논리장치, 줄여서 ALU 입니다.

4.3 산술논리장치

마이크로프로세서와같이연산을수행하는회로에서핵심컴포넌트중하나는 산술논리장치, 줄여서 ALU 입니다. 그림 4.5는 ALU 의기호를보여줍니다.

ALU 는그림에서 a 와 b 로표시된두개의데이터입력, 하나의함수입력 fn, 그리고 y 로표시된출력을가집니다. ALU 는 a 와 b 에대해연산을수행하고그결과를출력 y 에제공합니다. 입력 fn 은 a 와 b 에대한연산을선택합니다. 연산은보통덧셈이나뺄셈과같은산술연산몇가지와 and, or, xor 같은논리함수몇가지입니다. 그래서 ALU 라고불립니다.

ALU 는일반적으로상태요소가전혀없는조합회로입니다. ALU 는또한 0 이나부호와같은결과의속성을나타내는추가출력신호를가질수도있습니다.

다음코드는 2 비트 fn 신호로선택되는덧셈, 뺄셈, or, and 연산을지원하는, 16 비트입력과 16 비트출력을갖는 ALU 를보여줍니다.

```
class Alu extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(16.W))
    val b = Input(UInt(16.W))
    val fn = Input(UInt(2.W))
    val y = Output(UInt(16.W))
  })

  // some default value is needed
  io.y := 0.U

  // The ALU selection
```

```
  switch(io.fn) {
    is(0.U) { io.y := io.a + io.b }
    is(1.U) { io.y := io.a - io.b }
    is(2.U) { io.y := io.a | io.b }
    is(3.U) { io.y := io.a & io.b }
  }
}
```

이예제에서는새로운 Chisel 구문인 switch/is 구문을사용하여우리 ALU 의출력을선택하는표를기술합니다. 이유틸리티함수를사용하려면또다른 Chisel 패키지를임포트해야합니다.

```
import chisel3.util._
```

4.4 벌크연결

다중 IO 포트를가진컴포넌트를연결하기위해, Chisel 은벌크연결연산자 <> 를제공합니다. 이연산자는번들의각부분을양방향으로연결합니다. Chisel 은 연결을위해리프필드의이름을사용합니다. 이름이없으면연결되지않습니다.

예를들어, 파이프라인프로세서를만든다고가정해봅시다. fetch 단계는다음과같은인터페이스를가집니다:

```
class Fetch extends Module {
  val io = IO(new Bundle {
    val instr = Output(UInt(32.W))
    val pc = Output(UInt(32.W))
  })
  // ... Implementation of fetch
}
```

다음단계는 decode 단계입니다.

```
class Decode extends Module {
  val io = IO(new Bundle {
    val instr = Input(UInt(32.W))
    val pc = Input(UInt(32.W))
    val aluOp = Output(UInt(5.W))
    val regA = Output(UInt(32.W))
    val regB = Output(UInt(32.W))
```

```
  })
  // ... Implementation of decode
}
```

우리의단순한프로세서의마지막단계는 execute 단계입니다.

```
class Execute extends Module {
  val io = IO(new Bundle {
    val aluOp = Input(UInt(5.W))
    val regA = Input(UInt(32.W))
    val regB = Input(UInt(32.W))
    val result = Output(UInt(32.W))
  })
  // ... Implementation of execute
}
```

세단계를모두연결하려면단두개의 <> 연산자만있으면됩니다. 서브모듈의포트를부모모듈과연결할수도있습니다.

```
  val fetch = Module(new Fetch())
  val decode = Module(new Decode())
  val execute = Module(new Execute)

  fetch.io <> decode.io
  decode.io <> execute.io
  io <> execute.io
```

5 조합회로구성요소

이장에서는더복잡한시스템을구성하는데사용할수있는기본구성요소인다양한조합회로를살펴봅니다. 원칙적으로모든조합회로는불방정식으로기술할수있습니다. 하지만더흔하게는표형태의기술이더효율적입니다. 불방정식의추출과최소화는합성도구에맡깁니다. 표형태로기술하기에가장적합한두가지기본회로는디코더와인코더입니다.

5.1 조합회로

표준적인조합회로구성요소를몇가지설명하기에앞서, Chisel 에서조합회로를어떻게표현할수있는지살펴보겠습니다. 가장단순한형태는불표현식이며, 여기에이름을부여할수있습니다.

```
val e = (a & b) | c
```

불표현식은 Scala 값에대입함으로써이름 (e) 이부여됩니다. 이표현식은다른표현식에서재사용할수있습니다.

```
val f = ~e
```

이러한표현식은고정된것으로간주됩니다. = 로 e 에재대입을시도하면 Scala 컴파일러오류가발생합니다: reassignment to val. 아래와같이 Chisel 연산자 := 로시도하면,

```
e := c & b
```

런타임예외가발생합니다: Cannot reassign to read-only.

Chisel 은조건부갱신을통해조합회로를기술하는것도지원합니다. 이러한회로는 Wire 로선언됩니다. 그런다음 when 과같은조건부연산을사용하여회로의로직을기술합니다. 다음코드는 UInt 타입의 Wire w 를선언하고기본값 0 을대입합니다. when 블록은 Chisel Bool 을받아, cond 가 true.B 이면 w 에 3 을재대입합니다.

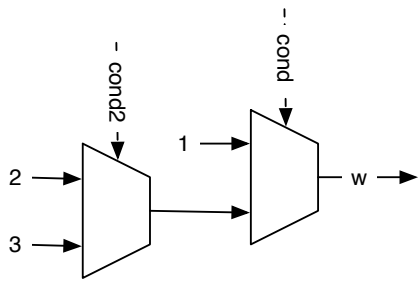


Figure 5.1: 멀티플렉서체인.

```
val w = Wire(UInt())

w := 0.U
when (cond) {
  w := 3.U
}
```

이회로의로직은멀티플렉서이며, 두입력은상수 0 과 3 이고선택신호는조건 cond 입니다. 우리가기술하는것은조건부실행이있는소프트웨어프로그램이아니라하드웨어회로라는점을유념하십시오.

Chisel 의조건구성요소 when 에는 그외의경우형태도있으며, 이는 .otherwise 라고부릅니다. 모든조건에서값을대입하도록하면기본값대입을생략할수있습니다.

```
val w = Wire(UInt())

when (cond) {
  w := 1.U
} .otherwise {
  w := 2.U
}
```

Chisel 은또한 .elsewhen 으로 (if/elseif/else 체인과같은) 조건문의체인도 지원합니다.

```
val w = Wire(UInt())
```

```
when (cond) {
  w := 1.U
} .elsewhen (cond2) {
  w := 2.U
} .otherwise {
  w := 3.U
}
```

이러한 when, .elsewhen, .otherwise 의체인은멀티플렉서체인을구성합니다. 그림 5.1은이멀티플렉서체인을보여줍니다. 이체인은우선순위를도입하는데, 즉 cond 가참이면나머지조건들은평가되지않습니다.

Scala 에서메서드를체인으로연결하는데필요한 .elsewhen 의 '.' 에주목하십시오. 이러한 .elsewhen 분기는임의로길게이어질수있습니다. 다만조건의체인이단일신호에의존한다면, 다음하위절에서디코더회로와함께소개할 switch 문을사용하는편이낫습니다.

더복잡한조합회로의경우, Wire 에기본값을대입하는것이실용적일수있습니다. 기본값대입은 WireDefault 로와이어선언과결합할수있습니다.

```
val w = WireDefault(0.U)

when (cond) {
  w := 3.U
}
// ... and some more complex conditional assignments
```

Scala 에 if, else if, else 가있는데왜 when, .elsewhen, .otherwise 를사용하는지의문이들수있습니다. 그러한 Scala 문들은 Scala 코드의조건부실행을위한것이지, Chisel(멀티플렉서) 하드웨어를생성하기위한것이아닙니다. 이러한 Scala 조건문은매개변수를받아조건에따라 서로다른하드웨어인스턴스를생성하는회로생성기를작성할때 Chisel 에서쓰임새가있습니다.

5.2 디코더

디코더는 n 비트의이진수를 m 비트신호로변환하며, 이때 $m \leq 2^n$ 입니다. 출력은원-핫으로인코딩됩니다 (정확히하나의비트만 1 입니다). 그림 5.2는 2 비트에서 4 비트로의디코더를보여줍니다. 표 5.1와같은진리표로디코더의기능을기술할수있습니다.

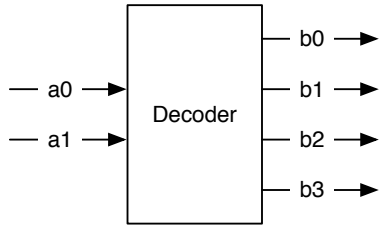


Figure 5.2: 2 비트에서 4 비트로의디코더입니다.

a	b
00	0001
01	0010
10	0100
11	1000

Table 5.1: 2 대 4 디코더의진리표입니다.

Chisel 의 switch 문은로직을진리표형태로기술합니다. switch 문을사용하려면 chisel.util 패키지를포함해야합니다.

```
import chisel3.util._
```

다음코드는 Chisel 의 switch 문을사용하여디코더를기술합니다:

```
  result := 0.U

  switch(sel) {
    is (0.U) { result := 1.U}
    is (1.U) { result := 2.U}
    is (2.U) { result := 4.U}
    is (3.U) { result := 8.U}
  }
```

위의 switch 문은 sel 신호가가질수있는모든값을나열하고, 디코딩된값을 result 신호에할당합니다. 가능한모든입력값을열거하더라도, Chisel 은여전히기본값을할당하도록요구한다는점에유의하십시오. 이는 result 에초기값 0 을할당함으로써수행한것입니다. 이할당은결코활성화되지않으므로합성도구에의해최적

화되어제거됩니다. 이는 VHDL 이나 Verilog 같은하드웨어기술언어에서의도치않은래치를발생시키는, 조합회로 (Chisel 에서는 Wire) 에대한불완전한할당상황을방지하기위한것입니다. Chisel 은불완전한할당을허용하지않습니다.

앞선예제에서는신호에부호없는정수를사용했습니다. 인코딩회로를더명확하게표현하려면이진표기법을사용할수도있습니다:

```
switch (sel) {
  is ("b00".U) { result := "b0001".U}
  is ("b01".U) { result := "b0010".U}
  is ("b10".U) { result := "b0100".U}
  is ("b11".U) { result := "b1000".U}
}
```

표는디코더기능을매우가독성있게표현하지만다소장황하기도합니다. 표를살펴보면규칙적인구조를발견할수있는데, 1 이 sel 로표현된숫자만큼왼쪽으로시프트됩니다. 따라서 Chisel 의시프트연산 « 로디코더를표현할수있습니다.

```
result := 1.U << sel
```

디코더는출력을멀티플렉서의데이터입력에대한 AND 게이트의인에이블로사용함으로써멀티플렉서의구성요소로사용됩니다. 그러나 Chisel 에서는코어라이브러리에 Mux 가제공되므로멀티플렉서를직접구성할필요가없습니다. 디코더는마이크로프로세서의주소버스일부비트에대한주소디코딩에도사용될수있습니다. 이출력들은마이크로프로세서에연결된메모리와다양한 IO 장치에대한선택신호로사용됩니다 (섹션 12.1 참고).

5.3 인코더

인코더는원-핫으로인코딩된입력신호를이진인코딩된출력신호로변환합니다. 인코더는디코더의역연산을수행합니다.

그림 5.3는 4 비트원-핫입력을 2 비트이진출력으로변환하는인코더를보여주며, 표 5.2는인코딩함수의진리표를보여줍니다. 그러나인코더는입력신호가원-핫으로인코딩되어있을때만예상대로동작합니다. 그외의모든입력값에대해서는출력이정의되지않습니다. 출력이정의되지않은함수는기술할수없으므로, 정의되지않은모든입력패턴을포착하는기본할당을사용합니다.

다음 Chisel 코드는기본값 0 을할당한다음, 유효한입력값에대해 switch 문을사용합니다.

```
b := "b00".U
```

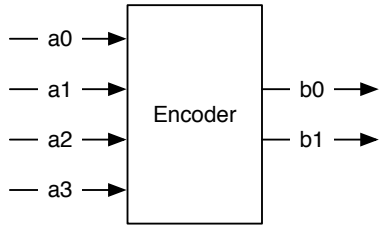


Figure 5.3: 4 비트에서 2 비트로의인코더입니다.

a	b
0001	00
0010	01
0100	10
1000	11
????	??

Table 5.2: 4 대 2 인코더의진리표입니다.

```
switch (a) {
  is ("b0001".U) { b := "b00".U}
  is ("b0010".U) { b := "b01".U}
  is ("b0100".U) { b := "b10".U}
  is ("b1000".U) { b := "b11".U}
}
```

디코더의경우, 로직을표현하는우아한한줄짜리문을찾아냈습니다. 이는또한 폭이넓은디코더를기술할수있게해줍니다. 그러나인코더에대해서는그러한표현식을알지못합니다.

더큰인코더를표현하려면 (간단한) 하드웨어생성기를작성해야합니다. 따라서 Scala 의루프구문을소개해야합니다. 다음두줄의 Scala 코드는 0 부터 9 까지세는루프를표현합니다.

```
// Loops i from 0 to 9
for (i <- 0 until 10) {
  // use i to index into a Wire or Vec
}
```

루프변수 i 는 Wire 나 Reg 의개별비트를, 또는 Vec 의한요소를인덱싱하는데사용될수있습니다. 이 Scala 생성기루프는하드웨어생성기를기술하는가장단순한형태입니다. 10 장에서는하드웨어생성기를작성하는방법을더자세히설명합니다. 이루프는회로생성시점에실행된다는점에유의하십시오. 이는하드웨어카운터가아닙니다.

인코더생성기에는 Vec 을사용하며, 각원소는인코더테이블의한열을나타냅니다. 다음코드는 16 비트인코더를보여주며, 출력은 4 비트폭입니다:

```
val v = Wire(Vec(16, UInt(4.W)))
v(0) := 0.U
for (i <- 1 until 16) {
  v(i) := Mux(hotIn(i), i.U, 0.U) | v(i - 1)
}
val encOut = v(15)
```

인코더의입력은 hotIn 이고출력은 encOut 입니다. Vec 원소 0 은기본경우 (0) 이며, 동시에 hotIn 에서최하위비트 (LSB) 가설정된경우의출력값도나타냅니다.

Vec 원소 1 부터 15 까지는멀티플렉서에연결됩니다. hotIn 의 i 위치비트가설정되어있으면멀티플렉서출력은해당인덱스이며, 그렇지않으면 0 입니다. 인코더가올바르게동작하려면입력신호가원-핫으로인코딩되어있다고가정합니다. 마지막으로모든벡터원소를하나의출력으로병합해야합니다. 벡터원소는입력의해당비트가 0 일때 0 이므로, 모든원소를 OR 함수로단순히결합할수있습니다. 루프에서는현재원소를바로앞벡터원소와 OR 연산합니다 (... | v(i-1)). 여러원소를함수로결합하는이러한연산을리듀스 (reduce) 라고도부릅니다. 따라서여기서는 OR 리덕션을수행하는것입니다.

5.4 아비터

우리는여러클라이언트가공유자원에대해보내는요청을중재하기위해아비터를사용합니다. 예를들어여러프로세서코어가하나의직렬포트 (UART) 를공유하는경우가있습니다.

그림 5.4는 4 비트아비터의회로도를보여줍니다. 이는네개의요청라인 (r0–r3) 과네개의승인라인 (g0–g3) 으로구성됩니다. 아비터는오직하나의요청만승인합니다. 예를들어요청입력이 0101 이면승인출력은 0001 이됩니다. 이아비터는낮은번호의입력에우선순위를둡니다. 따라서이를우선순위아비터라고부릅니다. 비트번호가낮을수록우선순위가높습니다.

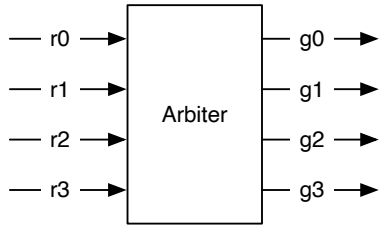


Figure 5.4: 4 비트아비터의기호입니다.

공정한아비터를만들기위해서는마지막중재를기억하는상태를추가해야합니다. 공정한아비터는 10.6.2절에서소개하겠습니다.

그림 5.5는 4 비트아비터의회로도를보여줍니다. 개별승인요청은더낮은비트가이미중재에서승리했는지확인해야합니다. 첫번째요청의경우, 승인 g0 은오직요청 r0 에만의존합니다. 두번째승인은요청 r1 이활성화되고요청 r0 이비활성화된경우에만중재에서승리할수있습니다. 이후의요청들에대해서도이러한조회가계속연쇄적으로이루어집니다.

다음코드는 3 개클라이언트를위한아비터를보여줍니다.

```
val grant = VecInit(false.B, false.B, false.B)
val notGranted = VecInit(false.B, false.B)

grant(0) := request(0)
notGranted(0) := !grant(0)
grant(1) := request(1) && notGranted(0)
notGranted(1) := !grant(1) && notGranted(0)
grant(2) := request(2) && notGranted(1)
```

이코드는그림 5.5의회로도와동일합니다 (다만여기서는 3 비트아비터에대한코드만보여줍니다). 요청, 승인, 미승인 (not-granted) 체인을나타내기위해 Bool 의벡터를사용합니다. grant(0) 이오직 request(0) 에만의존하는것을확인할수있습니다. notGranted 는더낮은비트의요청이아직승인되지않았다는정보를연쇄적으로전달하는데사용됩니다.

작은아비터는논리테이블로직접기술할수도있습니다. 다음코드는 3 비트아비터의테이블을보여줍니다.

```
val grant = WireDefault("b0000".U(3.W))
switch (request) {
  is ("b000".U) { grant := "b000".U}
```

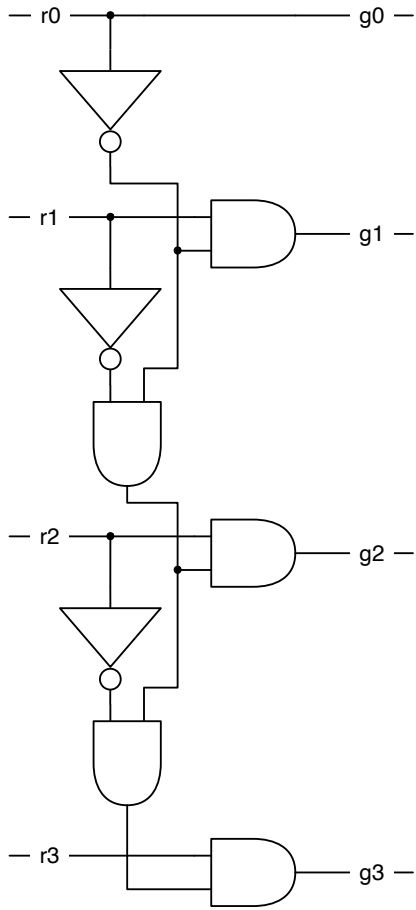


Figure 5.5: 4 비트아비터입니다.

```
    is ("b001".U) { grant := "b001".U}
    is ("b010".U) { grant := "b010".U}
    is ("b011".U) { grant := "b001".U}
    is ("b100".U) { grant := "b100".U}
    is ("b101".U) { grant := "b001".U}
    is ("b110".U) { grant := "b010".U}
    is ("b111".U) { grant := "b001".U}
  }
```

TODO: Chisel 에서*???* 를사용한패턴매칭도사용할수있을것입니다

그러나더큰중재회로에서는생성기루프로서 for 루프를사용하는, 새로배운기법을활용할것입니다. 다음코드는 n 개의요청과승인에대한매개변수화된아비터를보여줍니다. 여기서도마찬가지로 Bool 의 Vec 을사용합니다.

```
  val grant = VecInit.fill(n)(false.B)
  val notGranted = VecInit.fill(n)(false.B)

  grant(0) := request(0)
  notGranted(0) := !grant(0)
  for (i <- 1 until n) {
    grant(i) := request(i) && notGranted(i-1)
    notGranted(i) := !grant(i) && notGranted(i-1)
  }
```

위에보인코드는초기아비터버전의루프형태입니다. 이는 n 개의요청에대한중재회로를생성합니다. 수동버전 (언롤된루프) 과의작은차이점은마지막요청 (n - 1) 에대해서도 notGranted 와이어를생성한다는것입니다. 이와이어는사용되지않으며합성도구가이를최적화하여제거할것입니다.

5.5 우선순위인코더

원래의인코더설계에서는입력이원-핫으로인코딩되어있다고, 즉오직하나의비트만 1 일수있다고가정해야했습니다. 여러비트가설정된입력은유효하지않으며정의되지않은동작으로이어집니다.

이문제는인코더를중재회로와결합함으로써해결할수있으며, 이회로는설정된비트중가장높은우선순위의비트만선택합니다. 아비터의출력을인코더에입력하면우선순위인코더가만들어집니다. 그림 5.6는그회로도를보여줍니다.

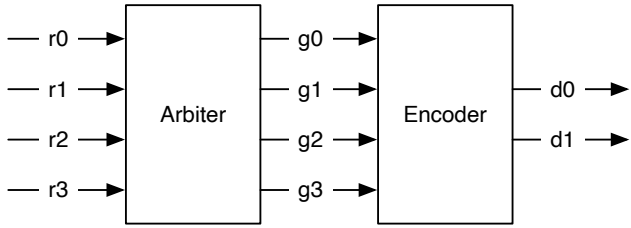


Figure 5.6: 아비터와인코더를결합하면우선순위인코더를만들수있습니다.

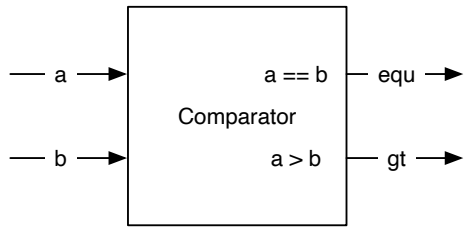


Figure 5.7: 간단한비교기입니다.

5.6 비교기

조합논리빌딩블록을다루는장의마지막회로로비교기를소개합니다. 그림 5.7는 비교기의개략도를보여줍니다. 이회로는다중비트입력두개를가지며이두값을비교합니다. 출력은두개로, (1) a 와 b 가같음을나타내는신호 (equ) 와 (2) a 가 b 보다큼을나타내는신호입니다. 이두출력만으로가능한모든비교를표현하기에충분합니다. 예를들어 equ 또는 gt 가참이면 $a >= b$ 조건이참임을알수있습니다. $a <= b$ 조건에대해서는 gt 의부정을검사합니다.

다음코드스니펫은비교기를보여줍니다. 보시다시피이는단두줄의 Chisel 코드에불과합니다. 따라서비교함수는보통다른컴포넌트에서직접사용되며모듈로감싸지않습니다.

```
val equ = a === b
val gt = a > b
```

5.7 연습문제

4 비트이진입력을 7 세그먼트디스플레이의인코딩으로변환하는조합회로를설계하십시오. 7 세그먼트디스플레이의초기용도였던십진숫자의코드를정의할수도있고, 추가로나머지비트패턴에대한인코딩을정의하여한자리숫자의 16 가지값을모두 16 진수로표시할수있게만들수도있습니다. 7 세그먼트디스플레이가있는 FPGA 보드가있다면, 스위치나버튼네개를회로의입력에연결하고출력을 7 세그먼트디스플레이에연결하십시오.

6 순차빌딩블록

순차회로는출력이입력과이전값에모두의존하는회로입니다. 우리는동기식설계(클록기반설계) 에관심이있으므로, 순차회로라고말할때는동기식순차회로를의미합니다.[1] 순차회로를구성하려면상태를저장할수있는요소, 즉이른바레지스터가필요합니다.

6.1 레지스터

순차회로를구성하는기본요소는레지스터입니다. 레지스터는 D 플립플롭의집합입니다. D 플립플롭은클록의상승에지에서입력값을포착하여출력에저장합니다. 다시말해, 레지스터는클록의상승에지에서입력값으로출력을갱신합니다.

그림 6.1는레지스터의개략도기호를보여줍니다. 이기호는입력 D 와출력 Q 를포함합니다. 각레지스터는또한 clock 신호를위한입력을포함합니다. 이전역클록신호는동기식회로내모든레지스터에연결되므로, 보통개략도에는그리지않습니다. 상자하단의작은삼각형은클록입력을상징하며이것이레지스터임을알려줍니다. 이후의개략도에서는클록신호를생략합니다. 전역클록신호의생략은 Chisel 에도반영되어있어, 레지스터의클록입력에신호를명시적으로연결할필

[1] 비동기논리와피드백을이용해순차회로를구성할수도있지만, 이는특수한주제이며 Chisel 로쉽게표현할수없습니다.

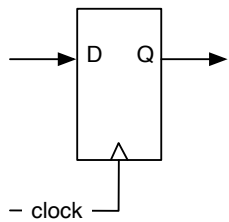


Figure 6.1: D 플립플롭기반레지스터입니다.

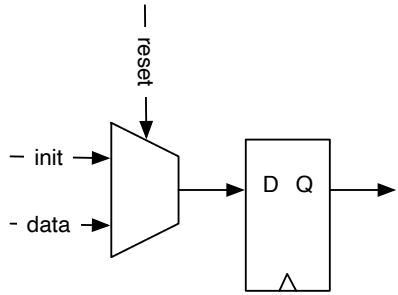


Figure 6.2: 동기식리셋을갖춘 D 플립플롭기반레지스터입니다.

요가없습니다. Chisel 에서입력 d 와출력 q 를가진레지스터는다음과같이정의합니다.

```
val q = RegNext(d)
```

레지스터에클록을연결할필요가없다는점에유의하십시오. Chisel 이이를암묵적으로처리합니다. 레지스터의입력과출력은벡터와번들의조합으로이루어진, 임의로복잡한타입일수있습니다.

레지스터는두단계로정의하고사용할수도있습니다.

```
val delayReg = Reg(UInt(4.W))

delayReg := delayIn
```

첫째, 레지스터를정의하고이름을부여합니다. 둘째, 신호 delayIn 을레지스터의입력에연결합니다. 또한레지스터의이름에 Reg 라는문자열이포함되어있음에유의하십시오. 조합회로와순차회로를쉽게구분하기위해, 이름에 Reg 표시를포함하는것이일반적인관행입니다.

레지스터는리셋시초기화될수있습니다. reset 신호는 clock 신호와마찬가지로 Chisel 에서암묵적으로존재합니다. 리셋값 (예: 0) 을레지스터생성자 RegInit 의매개변수로제공합니다. 레지스터의입력은 Chisel 대입문으로연결합니다.

```
val valReg = RegInit(0.U(4.W))

valReg := inVal
```

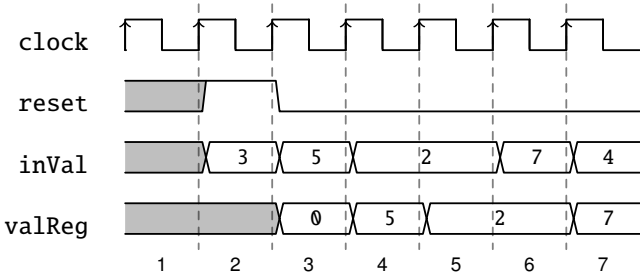


Figure 6.3: 리셋을갖춘레지스터의파형다이어그램입니다.

Chisel 의기본리셋구현방식은동기식리셋입니다.[2] 동기식리셋의경우 D 플립플롭자체에는변경이필요없으며, 대신리셋시의초기화값과데이터값사이를선택하는멀티플렉서를입력에추가해야합니다. 그림 6.2는리셋이멀티플렉서를구동하는동기식리셋을갖춘레지스터의개략도를보여줍니다. 하지만동기식리셋이매우흔히사용되므로, 현대의 FPGA 플립플롭에는멀티플렉서를위한 LUT 자원낭비를피하기위해플립플롭자체에동기식리셋 (및셋) 입력이내장되어있습니다.

순차회로는시간에따라값이변화합니다. 따라서그동작은시간에따른신호를보여주는다이어그램으로기술할수있습니다. 이러한다이어그램을파형또는 타이밍다이어그램이라고합니다.

그림 6.3는리셋과몇가지입력데이터가인가된레지스터의파형을보여줍니다. 시간은왼쪽에서오른쪽으로진행합니다. 그림상단에는회로를구동하는클록이보입니다. 첫번째클록사이클 (1) 에서는리셋이전이므로레지스터내용이정의되지않습니다. 두번째클록사이클에서는리셋이하이로인가되며, 이클록사이클의상승에지에서레지스터는초기값 0 을취합니다. 입력 inVal 은무시됩니다. 다음클록사이클에서는 reset 이 0 이되고, 다음상승에지에서 inVal 의값이캡처됩니다. 그이후로는마땅히그래야하듯 reset 이 0 으로유지되며, 레지스터출력은한클록사이클지연을두고입력신호를따라갑니다.

파형은회로의동작을그래픽으로명세하는데훌륭한도구입니다. 타이밍다이어그램은특히여러연산이병렬로일어나고데이터가회로를통해파이프라인방식으로이동하는더복잡한회로에서편리합니다. Chisel 테스터도테스트중에파형을생성할수있으며, 이는파형뷰어로표시하여디버깅에사용할수있습니다.

흔한설계패턴중하나는인에이블신호가있는레지스터입니다. 인에이블신호가 true(하이) 일때레지스터는입력을캡처하고, 그렇지않으면이전값을유지합니다.

[2] 비동기식리셋에대한지원도제공됩니다.

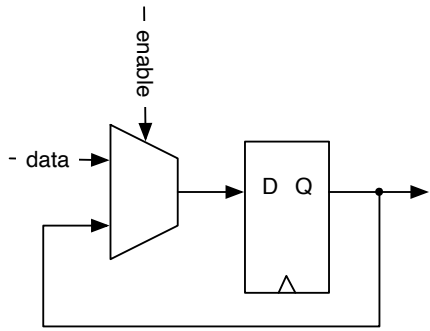


Figure 6.4: 인에이블신호가있는 D 플립플롭기반레지스터입니다.

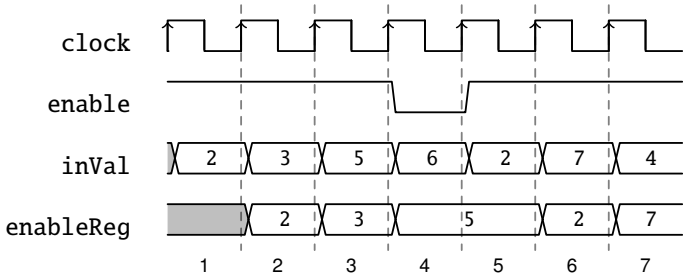


Figure 6.5: 인에이블신호가있는레지스터의파형다이어그램입니다.

인에이블은동기식리셋과유사하게레지스터입력단의멀티플렉서로구현할수있습니다. 멀티플렉서의한입력은레지스터출력의피드백입니다.

그림 6.4는인에이블신호가있는레지스터의회로도를보여줍니다. 이역시흔한설계패턴이기때문에, 현대의 FPGA 플립플롭은전용인에이블입력을갖추고있어레지스터인에이블을구현하는데추가 (LUT) 자원이필요하지않습니다.

그림 6.5는인에이블이있는레지스터의파형예시를보여줍니다. 대부분의시간동안 enable 은하이 (true) 이며레지스터는한클록사이클지연을두고입력을따라갑니다. 네번째클록사이클에서만 enable 이로우이며, 클록사이클 5 에서레지스터는값 (5) 을유지합니다.

인에이블이있는레지스터는조건부업데이트를사용하여몇줄의 Chisel 코드로기술할수있습니다.

```
val enableReg = Reg(UInt(4.W))

when (enable) {
  enableReg := inVal
}
```

레지스터에인에이블신호를사용하는것이매우흔하기때문에, Chisel 은두번째매개변수가인에이블신호인 RegEnable 을정의합니다.

```
val enableReg2 = RegEnable(inVal, enable)
```

인에이블이있는레지스터는리셋도가능합니다.

```
val resetEnableReg = RegInit(0.U(4.W))

when (enable) {
  resetEnableReg := inVal
}
```

리셋시레지스터초기화기능과인에이블기능은 3 개매개변수버전의 RegEnable 을사용하면결합할수있습니다. 첫번째매개변수는입력신호, 두번째매개변수는초기화값, 세번째매개변수는인에이블신호입니다.

```
val resetEnableReg2 = RegEnable(inVal, 0.U(4.W), enable)
```

레지스터는이름을부여하지않고도표현식의일부가될수있습니다. 다음회로는신호의현재값을지난클록사이클의값 (지연된값) 과비교하여신호의상승에지를감지합니다.

```
val risingEdge = din & !RegNext(din)
```

이제레지스터의기본적인활용법을모두살펴보았으니, 이러한레지스터를잘활용하여더흥미로운순차회로를만들어보겠습니다. 다음회로도부터는레지스터기호를더욱단순화하여입력의 D 와출력의 Q 를생략하겠습니다.

6.2 카운터

가장기본적인순차회로중하나는카운터입니다. 가장단순한형태로는, 카운터는출력이가산기에연결되고가산기의출력이레지스터의입력에연결된레지스터입니다. 그림 6.6는이러한자유실행카운터를보여줍니다.

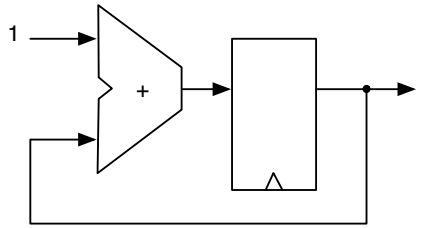


Figure 6.6: 가산기와레지스터가만나카운터가됩니다.

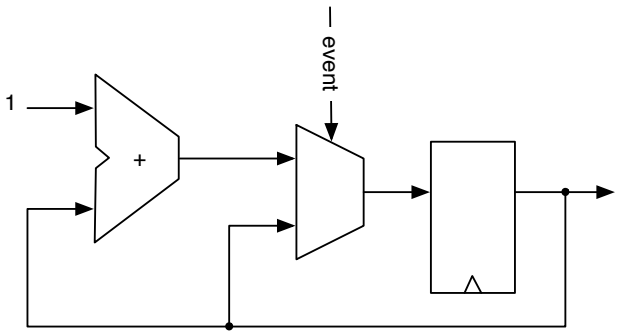


Figure 6.7: 이벤트계수하기.

4 비트레지스터를사용한자유실행카운터는 0 부터 15 까지세고나서다시 0 으로되돌아갑니다. 카운터는알려진값으로리셋될수도있어야합니다.

```
val cntReg = RegInit(0.U(4.W))

cntReg := cntReg + 1.U
```

이벤트를세고자할때는, 그림 6.7와다음코드에서보듯이조건을사용하여카운터를증가시킵니다.

```
val cntEventsReg = RegInit(0.U(4.W))
when(event) {
  cntEventsReg := cntEventsReg + 1.U
}
```

6.2.1 위로세기와아래로세기

어떤값까지세어올라간후 0 에서다시시작하려면, 예를들어 when 조건문을사용하여카운터값을최댓값상수 (다음예제에서는 N) 와비교해야합니다.

```
val cntReg = RegInit(0.U(8.W))

cntReg := cntReg + 1.U
when(cntReg === N) {
  cntReg := 0.U
}
```

우리는카운터에멀티플렉서를사용할수도있습니다.

```
val cntReg = RegInit(0.U(8.W))

cntReg := Mux(cntReg === N, 0.U, cntReg + 1.U)
```

아래로세고싶은경우에는, 먼저카운터레지스터를최댓값으로리셋하고 0 에도달했을때그값으로카운터를다시리셋합니다.

```
val cntReg = RegInit(N)

cntReg := cntReg - 1.U
when(cntReg === 0.U) {
  cntReg := N
}
```

코드를작성하면서더많은카운터를사용하게되면, 매개변수를가진함수를정의하여카운터를생성하도록할수있습니다.

```
// This function returns a counter
def genCounter(n: Int) = {
  val cntReg = RegInit(0.U(8.W))
  cntReg := Mux(cntReg === n.U, 0.U, cntReg + 1.U)
  cntReg
}

// now we can easily create many counters
val count10 = genCounter(10)
val count99 = genCounter(99)
```

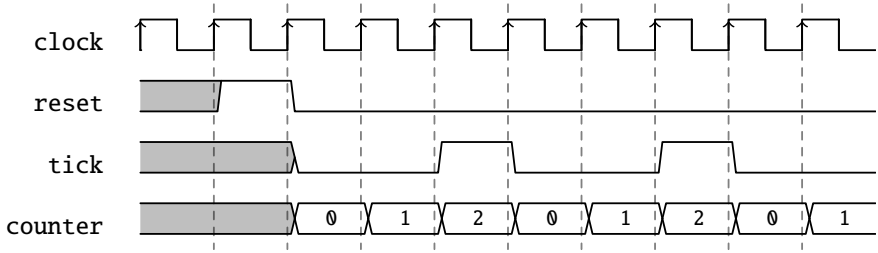


Figure 6.8: 느린주파수틱을생성하는파형도입니다.

함수 genCounter 의마지막문장은함수의반환값이며, 이예제에서는카운팅레지스터 cntReg 의출력입니다.

모든예제에서카운터의값이 0 부터 N 까지 (N 포함) 나타났다는점에유의하십시오. 클록사이클을 10 번세려면 N 을 9 로설정해야합니다. N 을 10 으로설정하면 오프바이원오류의전형적인예가될것입니다.

6.2.2 카운터로타이밍생성하기

이벤트를세는것외에도, 카운터는흔히시간 (벽시계시간으로서의시간) 이라는개념을생성하는데사용됩니다. 동기회로는고정된주파수를가진클록으로동작합니다. 회로는그러한클록틱단위로진행됩니다. 디지털회로에는클록틱을세는것외에시간이라는개념이존재하지않습니다. 클록주파수를알고있다면, Chisel "Hello World" 예제에서보듯이어떤주파수로 LED 를깜빡이는것과같이시간에맞춘이벤트를생성하는회로를만들수있습니다.

흔히쓰이는방법은회로에서필요한주파수 f_{tick} 를가진단일사이클 틱을생성하는것입니다. 그틱은 n 클록사이클마다발생하며, 여기서 $n = f_{clock}/f_{tick}$ 이고 틱은정확히한클록사이클길이입니다. 이틱은파생클록으로사용되는것이 아니라, 논리적으로주파수 f_{tick} 에서동작해야하는회로내레지스터들의인에이블신호로사용됩니다. 그림 6.8는 3 클록사이클마다생성되는틱의예를보여줍니다.

다음회로에서는 0 부터최댓값 N - 1 까지세는카운터를기술합니다. 최댓값에도달하면 tick 은한사이클동안 true 가되고, 카운터는 0 으로리셋됩니다. 0 부터 N - 1 까지셀때, N 클록사이클마다하나의논리적틱이생성됩니다.

```
val tickCounterReg = RegInit(0.U(32.W))
val tick = tickCounterReg === (N-1).U
```

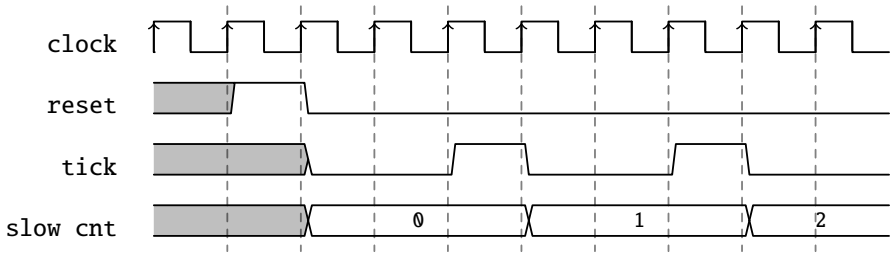


Figure 6.9: 느린주파수틱을사용하는예입니다.

```
tickCounterReg := tickCounterReg + 1.U
when (tick) {
  tickCounterReg := 0.U
}
```

n 클록사이클마다하나의틱이라는이논리적타이밍은, 이더느린논리적클록으로회로의다른부분을진행시키는데사용될수있습니다. 다음코드에서는 n 클록사이클마다 1 씩증가하는또다른카운터를사용합니다.

```
val lowFrequCntReg = RegInit(0.U(4.W))
when (tick) {
  lowFrequCntReg := lowFrequCntReg + 1.U
}
```

그림 6.9는틱의파형과매틱 (n 클록사이클) 마다증가하는느린카운터의파형을보여줍니다.

이더느린 논리적클록의활용예로는 LED 깜빡이기, 직렬버스의보레이트 (baud rate) 생성,[3] 7 세그먼트디스플레이멀티플렉싱을위한신호생성, 그리고버튼과스위치의디바운싱을위한입력값서브샘플링등이있습니다.

폭추론이레지스터의크기를정해주어야하지만, 레지스터정의시타입으로또는초기화값으로폭을명시적으로지정하는것이더좋습니다. 폭을명시적으로정의하면리셋값 0.U 가폭이 1 비트인카운터를만들어버리는뜻밖의상황을피할수있습니다.

[3] 보레이트는흔히초당비트수로표현되는정보전송속도의척도입니다. 이는송신부와수신부가서로동일해야합니다.

6.2.3 너드카운터

때로우리중일부는 너드가된듯한기분을느끼고싶어합니다. 예를들어, 카운터/틱생성의고도로최적화된버전을설계하고싶어질수있습니다. 표준카운터에는레지스터하나, 가산기 (또는감산기) 하나, 비교기하나라는자원이필요합니다. 레지스터나가산기에대해서는별로할수있는것이없습니다. 값을증가시키며세는경우, 비트문자열인어떤숫자와비교해야합니다. 이비교기는비트문자열에서 0 에해당하는자리에인버터를두고큰 AND 게이트를사용하여만들수있습니다. 0 까지감소시키며세는경우, 비교기는큰 NOR 게이트가되는데, 이는 ASIC 에서상수와비교하는비교기보다약간더저렴할수있습니다. 룩업테이블로논리가구성되는 FPGA 에서는 0 과비교하는것과 1 과비교하는것사이에차이가없습니다. 자원요구량은증가카운터와감소카운터모두동일합니다.

그러나영리한하드웨어설계자가부릴수있는트릭이하나더있습니다. 증가또는감소로세려면지금까지센모든비트와의비교가필요했습니다. N-2 부터 -1 까지세면어떨까요? 음수는최상위비트가 1 로설정되어있고, 양수는이비트가 0 으로설정되어있습니다. 카운터가 -1 에도달했음을감지하려면이비트만확인하면됩니다. 여기너드가만든카운터가있습니다.

```
val MAX = (N - 2).S(8.W)
val cntReg = RegInit(MAX)
io.tick := false.B

cntReg := cntReg - 1.S
when(cntReg(7)) {
  cntReg := MAX
  io.tick := true.B
}
```

6.2.4 타이머

우리가설계할수있는또다른형태의타이머는원샷타이머입니다. 원샷타이머는주방타이머와같습니다. 즉, 분수를설정하고시작을누릅니다. 지정된시간이경과하면알람이울립니다. 디지털타이머에는클록사이클단위의시간이로드됩니다. 그런다음 0 에도달할때까지감소하며셉니다. 0 이되면타이머는 완료을활성화합니다.

그림 6.10는타이머의블록다이어그램을보여줍니다. 레지스터는 load 를활성화하여 din 값으로로드될수있습니다. load 신호가비활성화되면감소카운팅이선택됩니다 (레지스터의입력으로 cntReg - 1 을선택하여). 카운터가 0 에도달하면

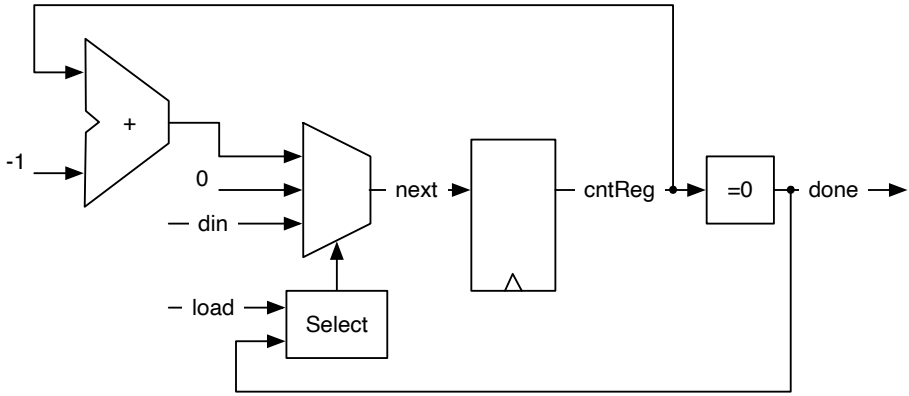


Figure 6.10: 원샷타이머입니다.

신호 done 이활성화되고, 멀티플렉서의 0 입력을선택함으로써카운터는카운팅을멈춥니다.

목록 6.1는타이머용 Chisel 코드를보여줍니다. 0 으로리셋되는 8 비트레지스터 cntReg 를사용합니다. 불리언값 done 은 cntReg 를 0 과비교한결과입니다. 입력멀티플렉서를위해, 기본값이 0 인와이어 next 를도입합니다. when/elsewhen 블록은선택함수로나머지두입력을도입합니다. 신호 load 는감소선택보다우선순위를가집니다. 마지막줄은 next 로표현되는멀티플렉서를레지스터 cntReg 의입력에연결합니다.

조금더간결한코드를지향한다면, 중간와이어 next 를사용하는대신멀티플렉서값을레지스터 reg 에직접할당할수있습니다.

6.2.5 펄스폭변조

펄스폭변조(PWM) 는일정한주기를가지며, 그주기내에서신호가 하이인시간을변조하는신호입니다.

그림 6.11는 PWM 신호를보여줍니다. 화살표는신호주기의시작지점을가리킵니다. 신호가 high 인시간의비율을듀티사이클이라고합니다. 처음두주기에서는듀티사이클이 25 % 이고, 다음두주기에서는 50 % 이며, 마지막두주기에서는 75 % 입니다. 펄스폭은 25 % 와 75 % 사이에서변조됩니다.

PWM 신호에 저역통과필터를추가하면간단한 디지털-아날로그변환기가만들어집니다. 저역통과필터는저항하나와커패시터하나만큼간단할수있습니다.

```
val cntReg = RegInit(0.U(8.W))
val done = cntReg === 0.U

val next = WireDefault(0.U)
when (load) {
  next := din
} .elsewhen (!done) {
  next := cntReg - 1.U
}
cntReg := next
```

Listing 6.1: A one-shot timer

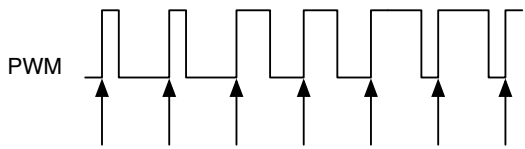


Figure 6.11: 펄스폭변조입니다.

다음코드예제는 10 클록사이클마다 3 클록사이클동안 high 인파형을생성합니다.

```
def pwm(nrCycles: Int, din: UInt) = {
  val cntReg =
      RegInit(0.U(unsignedBitLength(nrCycles-1).W))
  cntReg := Mux(cntReg === (nrCycles-1).U, 0.U, cntReg +
      1.U)
  din > cntReg
}

val din = 3.U
val dout = pwm(10, din)
```

재사용가능하고가벼운컴포넌트를제공하기위해 PWM 생성기에는함수를사용합니다. 이함수에는두개의매개변수가있습니다. 클록사이클수 (nrCycles) 로 PWM 을구성하는 Scala 정수와, PWM 출력신호의듀티사이클 (펄스폭) 을지정하는 Chisel 와이어 (din) 입니다. 이예제에서는카운터를표현하기위해멀티플렉서를사용합니다. 함수의마지막줄은카운터값을입력값 din 과비교하여 PWM 신호를반환합니다. Chisel 함수의마지막표현식이반환값이며, 이는비교함수에연결된와이어입니다.

n 까지 (포함) 부호없는숫자를표현하는데필요한카운터 cntReg 의비트수를지정하기위해함수 unsignedBitLength(n) 을사용합니다.[4] Chisel 에는숫자의부호있는표현을위한비트수를제공하는함수 signedBitLength 도있습니다.

PWM 신호의한가지응용은 LED 를조광하는것입니다. 이경우, 눈이저역통과필터역할을합니다. 위예제를확장하여삼각파함수로 PWM 생성을구동합니다. 그결과지속적으로밝기가변하는 LED 가만들어집니다.

```
val FREQ = 100000000 // a 100 MHz clock input
val MAX = FREQ/1000  // 1 kHz

val modulationReg = RegInit(0.U(32.W))

val upReg = RegInit(true.B)

when (modulationReg < FREQ.U && upReg) {
  modulationReg := modulationReg + 1.U
} .elsewhen (modulationReg === FREQ.U && upReg) {
  upReg := false.B
```

[4] 부호없는숫자 n 을이진수로표현하는데필요한비트수는 $\lfloor log_2(n) \rfloor + 1$ 입니다.

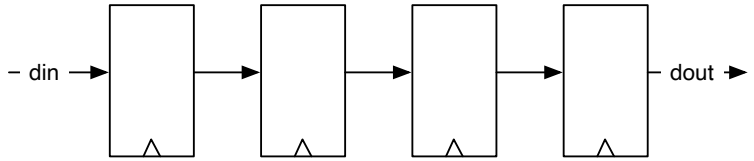


Figure 6.12: 4 단시프트레지스터입니다.

```
} .elsewhen (modulationReg > 0.U && !upReg) {
  modulationReg := modulationReg - 1.U
} .otherwise { // 0
  upReg := true.B
}

// divide modReg by 1024 (about the 1 kHz)
val sig = pwm(MAX, modulationReg >> 10)
```

변조를위해두개의레지스터를사용합니다. (1) 증가와감소를세기위한 modulationReg 와 (2) 증가로셀지감소로셀지를결정하는플래그로서의 upReg 입니다. 예제에서는클록입력의주파수 (100 MHz) 까지증가시키며세는데, 이는 0.5 Hz 의신호를만들어냅니다. 길게이어지는 when/.elsewhen/.otherwise 표현식은증가또는감소카운팅과방향전환을처리합니다.

우리 PWM 은 1 kHz 신호를생성하기위해주파수의 1000 분의 1 까지만 카운트하므로, 변조신호를 1000 으로나누어야합니다. 실제나눗셈은하드웨어에서매우비용이크므로, 단순히오른쪽으로 10 비트시프트하는데, 이는 $2^{10} = 1024$ 로나누는것과같습니다. PWM 회로를함수로정의했으므로, 함수호출로해당회로를간단히인스턴스화할수있습니다. 와이어 sig 는변조된 PWM 신호를나타냅니다.

6.3 시프트레지스터

시프트레지스터는순서대로연결된플립플롭들의모음입니다. 각플립플롭의출력은다음플립플롭의입력에연결됩니다. 그림 6.12는 4 단시프트레지스터를보여줍니다. 이회로는매클록틱마다데이터를왼쪽에서오른쪽으로 시프트합니다. 이단순한형태에서회로는 din 에서 dout 까지 4 사이클지연을구현합니다.

이단순한시프트레지스터의 Chisel 코드는 (1) 4 비트레지스터 shiftReg 를생성하고, (2) 시프트레지스터의하위 3 비트를입력 din 과연결하여레지스터의

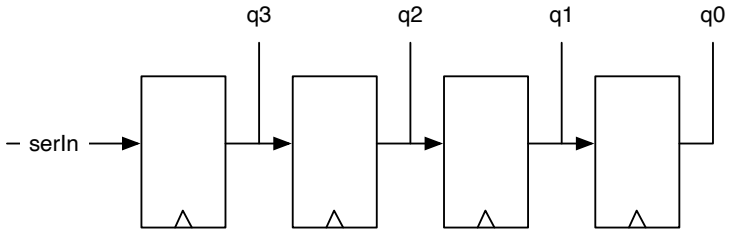


Figure 6.13: 병렬출력을갖는 4 비트시프트레지스터입니다.

다음입력으로사용하며, (3) 레지스터의최상위비트 (MSB) 를출력 dout 으로사용합니다.

```
val shiftReg = Reg(UInt(4.W))
shiftReg := shiftReg(2, 0) ## din
val dout = shiftReg(3)
```

시프트레지스터는직렬데이터를병렬데이터로변환하거나병렬데이터를직렬데이터로변환하는데자주사용됩니다. 11.2 절은수신및송신기능에시프트레지스터를사용하는직렬포트를보여줍니다.

6.3.1 병렬출력을갖는시프트레지스터

직렬입력-병렬출력구성의시프트레지스터는직렬입력스트림을병렬워드로변환합니다. 이는직렬포트 (UART) 의수신기능에사용될수있습니다. 그림 6.13는 4 비트시프트레지스터를보여주는데, 각플립플롭출력이하나의출력비트에연결되어있습니다. 4 클록사이클후, 이회로는 4 비트직렬데이터워드를 q 에서사용가능한 4 비트병렬데이터워드로변환합니다. 이예제에서는비트 0(최하위비트)이먼저전송되므로, 전체워드를읽고자할때마지막단계에도착한다고가정합니다.

다음 Chisel 코드에서는시프트레지스터 outReg 를 0 으로초기화합니다. 그런다음 MSB 로부터시프트인하는데, 이는오른쪽시프트를의미합니다. 병렬결과인 q 는단순히레지스터 outReg 를읽은값입니다.

```
val outReg = RegInit(0.U(4.W))
outReg := serIn ## outReg(3, 1)
val q = outReg
```

그림 6.13는병렬출력기능을갖는 4 비트시프트레지스터를보여줍니다.

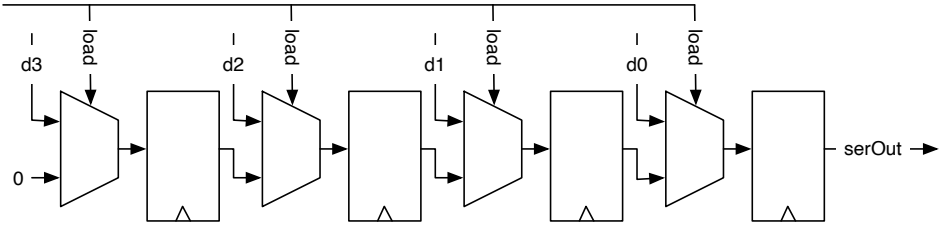


Figure 6.14: 병렬로드를갖는 4 비트시프트레지스터입니다.

6.3.2 병렬로드를갖는시프트레지스터

병렬입력-직렬출력구성의시프트레지스터는워드 (바이트) 의병렬입력스트림을직렬출력스트림으로변환합니다. 이는직렬포트 (UART) 의송신기능에사용될수있습니다.

그림 6.14는병렬로드기능을갖는 4 비트시프트레지스터를보여줍니다. 해당기능의 Chisel 서술은비교적간단합니다.

```
val loadReg = RegInit(0.U(4.W))
when (load) {
  loadReg := d
} otherwise  {
  loadReg := 0.U ## loadReg(3, 1)
}
val serOut = loadReg(0)
```

이제오른쪽으로시프트하면서 MSB 에 0 을채워넣는다는점에유의하십시오.

6.4 메모리

메모리는레지스터들의모음, 즉 Chisel 에서는 Vec 의 Reg 로구성할수있습니다. 그러나이는하드웨어에서비용이크며, 더큰메모리구조는 SRAM으로구축됩니다. ASIC 의경우, 메모리컴파일러가메모리를구성합니다. FPGA 는온칩메모리블록을포함하는데, 이를블록램 (block RAM) 이라고도부릅니다. 이러한온칩메모리블록들은더큰메모리를위해결합될수있습니다. FPGA 의메모리는보통하나의읽기포트와하나의쓰기포트를갖거나, 실행중에읽기와쓰기사이를전환할수있는두개의포트를갖습니다.

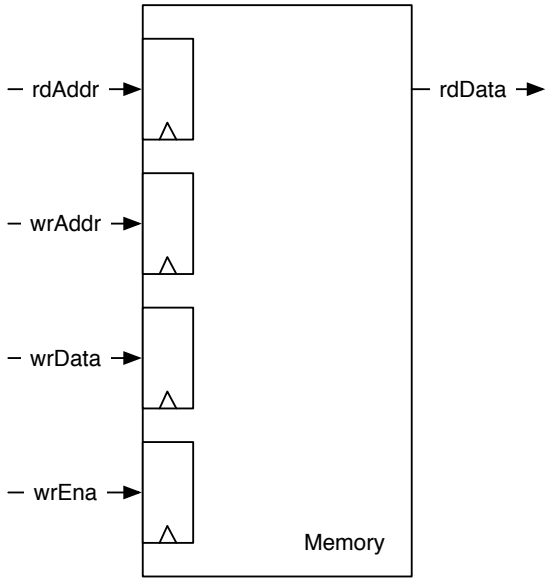


Figure 6.15: 동기메모리입니다.

FPGA(그리고 ASIC 도마찬가지로) 는보통동기메모리를지원합니다. 동기메모리는입력 (읽기및쓰기주소, 쓰기데이터, 쓰기인에이블) 에레지스터를갖습니다. 이는주소를설정한후한클록사이클뒤에읽기데이터를사용할수있음을의미합니다.

그림 6.15는이러한동기메모리의개략도를보여줍니다. 이메모리는하나의읽기포트와하나의쓰기포트를갖는듀얼포트방식입니다. 읽기포트는하나의입력, 즉읽기주소 (rdAddr) 와하나의출력, 즉읽기데이터 (rdData) 를갖습니다. 쓰기포트는세개의입력을갖는데, 주소 (wrAddr), 기록할데이터 (wrData), 그리고쓰기인에이블 (wrEna) 입니다. 모든입력에대해메모리내부에레지스터가있어동기적동작을나타낸다는점에유의하십시오.

온칩메모리를지원하기위해 Chisel 은메모리생성자 SyncReadMem 을제공합니다. 목록 6.2는바이트단위입출력데이터와쓰기인에이블을갖춘 1 KiB 메모리를구현하는 Memory 컴포넌트를보여줍니다.

흥미로운질문은동일한클록사이클에서읽어나오는것과같은주소에새로운값이쓰여질때, 읽기에서어떤값이반환되는가하는것입니다. 우리는메모리의 read-during-write 동작에관심이있습니다. 세가지가능성이있습니다: 새로

```
class Memory() extends Module {
  val io = IO(new Bundle {
    val rdAddr = Input(UInt(10.W))
    val rdData = Output(UInt(8.W))
    val wrAddr = Input(UInt(10.W))
    val wrData = Input(UInt(8.W))
    val wrEna = Input(Bool())
  })

  val mem = SyncReadMem(1024, UInt(8.W))

  io.rdData := mem.read(io.rdAddr)

  when(io.wrEna) {
    mem.write(io.wrAddr, io.wrData)
  }
}
```

Listing 6.2: 1 KiB 의동기식메모리.

쓰인값, 이전값, 또는정의되지않은값 (이전값의일부비트와새로쓰인데이터의일부비트가섞여있을수있음) 입니다. FPGA 에서어떤가능성이제공되는지는 FPGA 종류에따라다르며, 때로는명시적으로지정할수있습니다. Chisel 문서에따르면읽기데이터는정의되지않은것으로되어있습니다.

새로쓰인값을읽어내고자한다면, 주소가같음을감지하여쓰기데이터를 포워딩하는포워딩회로를구성할수있습니다. 그림 6.16은포워딩회로를갖춘메모리를보여줍니다. 읽기주소와쓰기주소를비교하고쓰기인에이블로게이팅하여, 쓰기데이터의포워딩경로와메모리읽기데이터중하나를선택합니다. 쓰기데이터는레지스터를통해한클록사이클만큼지연됩니다.

목록 6.3는포워딩회로를포함한동기식메모리에대한 Chisel 코드를보여줍니다. 다음클록사이클에서도읽기값을제공하는동기식메모리에맞추기위해, 쓰인데이터를다음클록사이클에사용할수있도록레지스터 (wrDataReg) 에저장해야합니다. 두입력주소 (wrAddr 과 rdAddr) 를비교하고, wrEna 가참인지확인하여포워딩조건으로삼습니다. 그조건역시한클록사이클만큼지연됩니다. 멀티플렉서가포워딩 (쓰기) 데이터와메모리에서의읽기데이터중하나를선택합니다.

이패턴이흔하기때문에, Chisel 은 SyncReadMem 에 read-during-write 동작을정의하는선택적매개변수를제공합니다. WriteFirst 는필요한경우포워딩

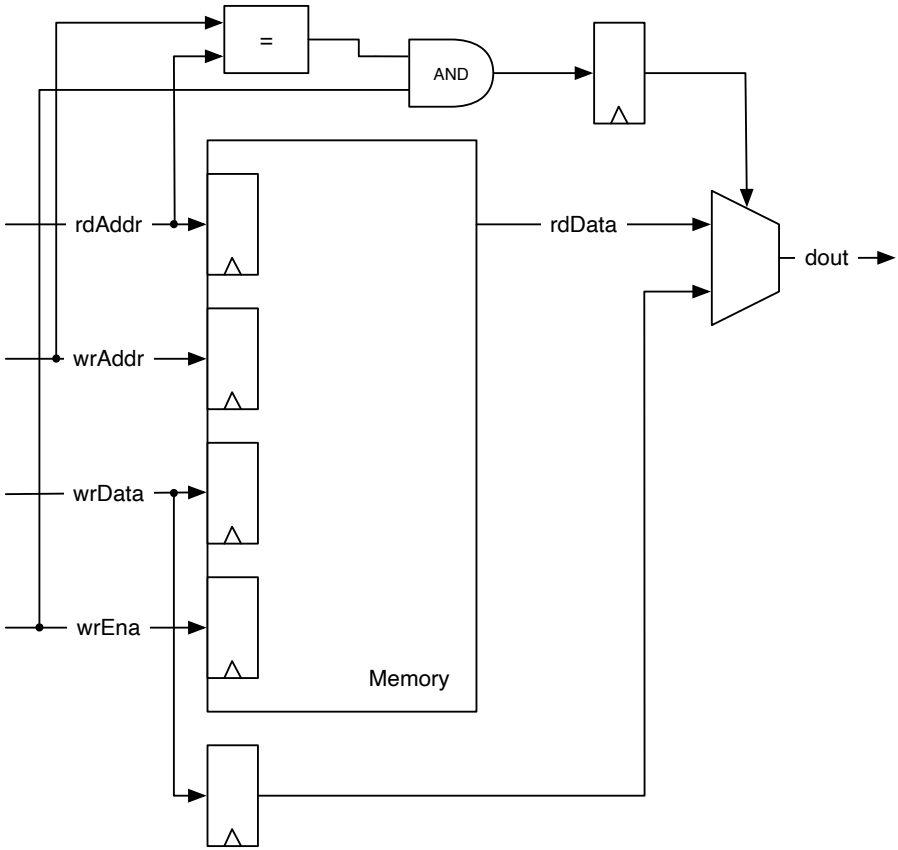


Figure 6.16: 정의된 read-during-write 동작을위한포워딩이포함된동기식 메모리입니다.

```
class ForwardingMemory() extends Module {
  val io = IO(new Bundle {
    val rdAddr = Input(UInt(10.W))
    val rdData = Output(UInt(8.W))
    val wrAddr = Input(UInt(10.W))
    val wrData = Input(UInt(8.W))
    val wrEna = Input(Bool())
  })

  val mem = SyncReadMem(1024, UInt(8.W))

  val wrDataReg = RegNext(io.wrData)
  val doForwardReg = RegNext(io.wrAddr === io.rdAddr &&
      io.wrEna)

  val memData = mem.read(io.rdAddr)

  when(io.wrEna) {
    mem.write(io.wrAddr, io.wrData)
  }

  io.rdData := Mux(doForwardReg, wrDataReg, memData)
}
```

Listing 6.3: 포워딩회로를갖춘메모리.

```
val hello = "Hello, World!"
val helloHex =
    hello.map(_.toInt.toHexString).mkString("\n")
val file = new java.io.PrintWriter("hello.hex")
file.write(helloHex)
file.close()

val mem = SyncReadMem(1024, UInt(8.W))
loadMemoryFromFileInline(mem, "hello.hex",
    firrtl.annotations.MemoryLoadFileType.Hex)
```

Listing 6.4: 메모리초기화.

을포함한 Verilog 코드를생성합니다. 나머지두옵션은 ReadFirst 와 Undefined 입니다.

```
val mem = SyncReadMem(1024, UInt(8.W),
    SyncReadMem.WriteFirst)
```

FPGA 의메모리는이진또는 16 진초기화파일로초기화할수있습니다. 이파일들은해당메모리의항목수와같은줄수를갖는단순한 ASCII 텍스트파일입니다. 각문자는한비트또는네비트를나타냅니다. 전통적으로이진파일은 .bin 파일확장자를사용하고, 16 진파일은 .hex 를사용합니다. loadMemoryFromFile 을사용하면별도의 Verilog 파일이생성되며, ChiselTest 에서도동작합니다. 초기화는 readmemb 또는 readmemh 호출을기반으로합니다.

생성시점에 Scala 에서온칩메모리를초기화하려면, 먼저내용을파일에기록한후 loadMemoryFromFile 을사용해야합니다. 목록 6.4는문자열로메모리를초기화하는예시를보여줍니다.

Chisel 은동기식쓰기와비동기식읽기를갖는메모리를나타내는 Mem 도제공합니다. 이메모리유형은보통 FPGA 에서직접제공되지않으므로, 합성도구는이를플립플롭으로구성하게됩니다. 따라서 SyncReadMem 을사용하는것을권장합니다. 비동기식읽기동작이필요하고사용중인 FPGA 에서해당자원을이용할수있는경우 (예: Xilinx FPGA 의 LUT RAM), 이를 BlackBox 로직접구현할수있습니다. 벤더는일반적으로이를위해그대로사용할수있는코드템플릿을제공합니다.

6.5 연습문제

지난연습문제의 7-세그먼트인코더를사용하여, 디스플레이를 0 부터 F 까지전환하는입력으로 4 비트카운터를추가하십시오. 이카운터를 FPGA 보드의클록에직접연결하면 16 개의숫자가모두겹쳐보이게됩니다 (7 개세그먼트가모두켜지게됩니다). 따라서카운트속도를늦출필요가있습니다. 500 밀리초마다한사이클동안의 틱신호를생성할수있는또다른카운터를만드십시오. 그신호를 4 비트카운터의인에이블신호로사용하십시오.

생성기함수로 PWM 파형을구성하고, 함수 (삼각파또는사인파함수) 로임계값을설정하십시오. 삼각파함수는카운트를올리고내리는방식으로만들수있습니다. 사인파함수는 Scala 코드몇줄로생성할수있는룩업테이블로만들수있습니다 (10.3 절참조). FPGA 보드의 LED 를그변조된 PWM 함수로구동하십시오. PWM 신호는어떤주파수로설정해야합니까? 드라이버는어떤주파수로동작하고있습니까?

디지털설계는흔히종이위에회로로스케치됩니다. 모든세부사항을나타낼필요는없습니다. 이책의그림들처럼블록다이어그램을사용합니다. 회로의개략적표현과 Chisel 서술사이를능숙하게오가며옮길수있는것은중요한능력입니다. 다음회로들에대한블록다이어그램을스케치하십시오:

```
val dout = WireDefault(0.U)

switch(sel) {
  is(0.U) { dout := 0.U }
  is(1.U) { dout := 11.U }
  is(2.U) { dout := 22.U }
  is(3.U) { dout := 33.U }
  is(4.U) { dout := 44.U }
  is(5.U) { dout := 55.U }
}
```

다음은레지스터를포함하는다소더복잡한회로입니다:

```
val regAcc = RegInit(0.U(8.W))

switch(sel) {
  is(0.U) { regAcc := regAcc}
  is(1.U) { regAcc := 0.U}
  is(2.U) { regAcc := regAcc + din}
  is(3.U) { regAcc := regAcc - din}
}
```

심화연습문제로, 온칩메모리를사용해보십시오. SyncReadMem 을인스턴스화하고서로통신하는두개의상태기계를만드십시오. 첫번째상태기계는문자열을메모리에씁니다. 완료되면두번째상태기계를시작시켜메모리에서그문자열을다시읽어옵니다. 읽는상태기계에서간단한 printf 문으로테스트하십시오. 읽을때읽기값이메모리에주소를적용한시점보다한클록사이클늦게나온다는점에유의하십시오. 이예제를확장하여메모리를테스트하는 Chisel 테스트를작성할수있습니다.

때로는노트북에서메모리의내용을다운로드하고싶을때가있습니다. 예를들어프로그램을로드하기위해프로세서를만들때가그렇습니다. FPGA 보드와노트북을연결하는직렬포트가있다고가정합시다 (11.2 절참조). 구성을마친후메모리내용을 FPGA 로다운로드하기위해, FPGA 내상태기계를활용해직렬포트상에서동작하는프로토콜을구상할수있습니까? 실행시점에다운로드하면메모리내용이바뀔때마다 FPGA 를다시합성하는것도피할수있습니다.

7 입력처리

외부세계로부터우리의동기식회로로들어오는입력신호는대개클록에동기화되어있지않으며, 비동기적입니다. 입력신호는 0 에서 1 로, 또는 1 에서 0 으로깔끔하게전이하지않는소스에서올수있습니다. 그예로바운싱하는버튼이나스위치가있습니다. 입력신호는우리의동기식회로에서전이를유발할수있는스파이크로인해잡음이섞일수있습니다. 이장에서는이러한입력조건을다루는회로에관해설명합니다.

후자의두문제, 즉스위치디바운싱과잡음필터링은외부의아날로그컴포넌트로도해결할수있습니다. 그러나이러한문제를디지털영역에서다루는것이비용효율적입니다.

7.1 비동기입력

시스템클록에동기화되어있지않은입력신호를비동기신호라고합니다. 이러한신호는플립플롭입력의셋업시간과홀드시간을위반할수있습니다. 이러한위반은플립플롭의 준안정성을초래할수있습니다. 준안정성은 0 과 1 사이의출력값을초래하거나, 발진을초래할수있습니다. 그러나일정시간이지나면플립플롭은 0 또는 1 로안정화됩니다.

외부의비동기입력신호에서또다른흔한문제는해당신호가상승클록에지근처에서변화하며회로의두곳이상에서사용되는경우입니다. 서로다른지연시간으로인해, 해당입력의서로다른사용처들은서로다른클록사이클에등록될수있으며, 이는일부가정을위반할수있습니다.[1]

우리는준안정성을피할수는없지만, 그영향을억제할수는있습니다. 고전적인해결책은입력단에두개의플립플롭을사용하는것입니다. 이방식의전제는, 첫번째플립플롭이준안정상태가되더라도클록주기내에안정상태로수렴하여두번째플립플롭의셋업시간과홀드시간이위반되지않는다는것입니다.

그림 7.1는외부세계와동기식회로사이의경계를보여줍니다. 입력동기화기는두개의플립플롭으로구성됩니다. 입력동기화기를위한 Chisel 코드는두개의레지스터를인스턴스화하는한줄짜리코드입니다.

[1] 저는이문제를한번겪었으며, 오류를찾는데상당한시간이걸렸습니다.

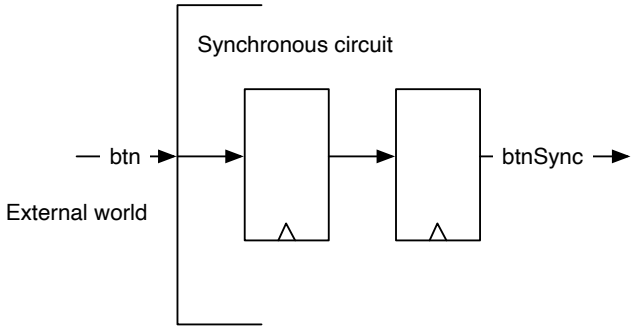


Figure 7.1: 입력동기화기.

```
val btnSync = RegNext(RegNext(btn))
```

모든비동기외부신호에는입력동기화기가필요합니다.[2] 또한전역리셋이라고도불리는외부리셋신호도동기화해야합니다. 리셋신호는회로내다른플립플롭의리셋신호로사용되기전에두개의플립플롭을거쳐야합니다. 구체적으로, 리셋의해제 (deassertion) 는클록에동기화되어야합니다.

7.2 디바운싱

스위치와버튼은켜짐과꺼짐사이를전이하는데어느정도시간이필요할수있습니다. 전이도중에스위치는두상태사이를바운싱할수있습니다. 이러한신호를추가처리없이사용하면, 우리가원하는것보다더많은전이이벤트를감지하게될수있습니다. 한가지해결책은시간을이용해이러한바운싱을걸러내는것입니다. 최대바운싱시간을 t_{bounce} 라고가정할때, 우리는주기 $T > t_{bounce}$ 로입력신호를샘플링할것입니다. 이후단계에서는샘플링된신호만을사용할것입니다.

이렇게긴주기로입력을샘플링할때, 0 에서 1 로의전이시오직하나의샘플만이바운싱구간에속할수있음을알수있습니다. 그이전샘플은안전하게 0 을읽을것이고, 바운싱구간이후의샘플은안전하게 1 을읽을것입니다. 바운싱구간에속한샘플은 0 일수도 1 일수도있습니다. 그러나이는문제가되지않는데, 그샘플은여전히 0 인샘플들에속하거나이미 1 인샘플에속하게되기때문입니다. 핵심은우리가 0 에서 1 로의전이를오직한번만갖는다는점입니다.

[2] 예외는입력신호가동기식출력신호에종속되어있고최대전파지연을알고있는경우입니다. 고전적인예로는마이크로프로세서등에의한비동기 SRAM 과동기식회로간의인터페이싱이있습니다.

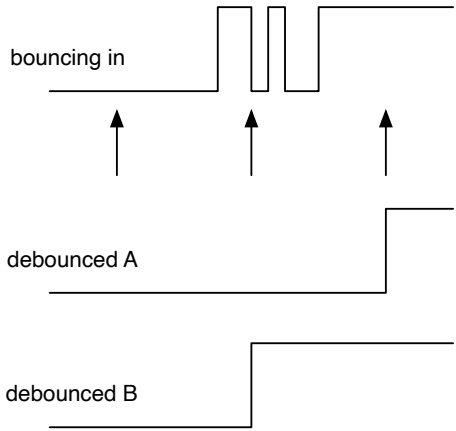


Figure 7.2: 입력신호의디바운싱.

그림 7.2는디바운싱을위한샘플링이실제로동작하는모습을보여줍니다. 위쪽 신호는바운싱하는입력을나타내고, 아래쪽화살표는샘플링시점을나타냅니다. 이샘플링시점사이의간격은최대바운싱시간보다길어야합니다. 첫번째샘플은안전하게 0 을샘플링하며, 그림의마지막샘플은 1 을샘플링합니다. 가운데샘플은 바운싱시간에속합니다. 이는 0 일수도 1 일수도있습니다. 두가지가능한결과는 debounced A 와 debounced B 로표시되어있습니다. 둘다 0 에서 1 로의단일 전이를가집니다. 이두결과사이의유일한차이는버전 A 의전이가한샘플주기만큼더늦다는점입니다. 그러나이는대개문제가되지않습니다.

디바운싱을위한 Chisel 코드는동기화기의코드보다조금더발전된형태입니다. 우리는 6.2.2 절에서했던것처럼, 단일사이클 tick 신호를전달하는카운터로 샘플타이밍을생성합니다.

```
val fac = 100000000/100


val btnDebReg = RegInit(false.B)

val cntReg = RegInit(0.U(32.W))
val tick = cntReg === (fac-1).U

cntReg := cntReg + 1.U
when (tick) {
```

```
    cntReg := 0.U
    btnDebReg := btnSync
  }
```

먼저샘플링주파수를결정해야합니다. 위예시는 100 MHz 클록을가정하며, (바운싱시간이 10 ms 미만이라고가정할때) 100 Hz 의샘플링주파수를결과로얻습니다. 카운터의최대값은분주계수인 fac 입니다. 우리는디바운싱된신호를위해리셋값이없는레지스터 btnDebReg 를정의합니다. 레지스터 cntReg 는카운터역할을하며, 카운터가최대값에도달했을때 tick 신호가참이됩니다. 이경우, when 조건이 true 가되어: (1) 카운터가 0 으로리셋되고 (2) 디바운스레지스터가입력샘플을저장합니다. 우리의예시에서, 입력신호는앞절에서보여준입력동기화기의출력으로서 btnSync 라는이름을가집니다.

디바운싱회로는동기화기회로다음에옵니다. 먼저비동기신호를동기화해야하며, 그이후에디지털영역에서추가로처리할수있습니다.

7.3 입력신호의필터링

때로는우리의입력신호에잡음이섞여있을수있으며, 입력동기화기와디바운싱유닛으로의도치않게샘플링될수있는스파이크가포함되어있을수있습니다. 이러한입력스파이크를필터링하는한가지방법은다수결투표회로를사용하는것입니다. 가장단순한경우, 세개의샘플을취해다수결투표를수행합니다. 중앙값함수와관련된 다수결함수는다수의값을결과로산출합니다. 디바운싱을위해샘플링을사용하는우리의경우, 샘플링된신호에대해다수결투표를수행합니다. 다수결투표는신호가샘플링주기보다더오래안정적으로유지되도록보장합니다.

그림 7.3는다수결투표회로를보여줍니다. 이회로는디바운싱샘플링에사용했던 tick 신호로활성화되는 3 비트시프트레지스터로구성됩니다. 세레지스터의출력은다수결투표회로로입력됩니다. 다수결투표함수는샘플주기보다짧은모든신호변화를걸러냅니다.

다음의 Chisel 코드는 tick 신호로활성화되는 3 비트시프트레지스터와투표함수를보여주며, 이는신호 btnClean 을결과로산출합니다.

다수결투표는매우드물게필요하다는점에유의하십시오.

```
val shiftReg = RegInit(0.U(3.W))
when (tick) {
  // shift left and input in LSB
  shiftReg := shiftReg(1, 0) ## btnDebReg
}
// Majority voting
```

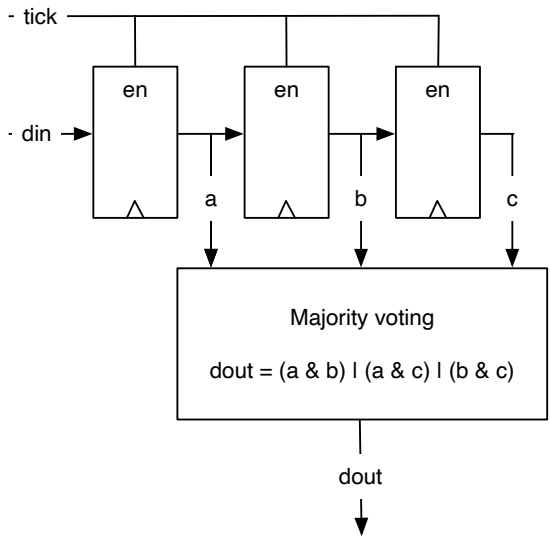


Figure 7.3: 샘플링된입력신호에대한다수결투표.

```
val btnClean = (shiftReg(2) & shiftReg(1)) | (shiftReg(2)
    & shiftReg(0)) | (shiftReg(1) & shiftReg(0))
```

신중하게처리된입력신호의출력을사용하려면, 먼저 RegNext 지연소자로상승에지를검출한다음이신호를 btnClean 의현재값과비교합니다. 이예제에서는단일사이클 risingEdge 신호를사용하여카운터를증가시킵니다.

```
val risingEdge = btnClean & !RegNext(btnClean)

// Use the rising edge of the debounced and
// filtered button to count up
val reg = RegInit(0.U(8.W))
when (risingEdge) {
  reg := reg + 1.U
}
```

7.4 함수로입력처리결합하기

입력처리과정을요약하기위해, Chisel 코드를몇가지더살펴봅니다. 제시된회로는작지만재사용가능한빌딩블록이므로, 이를함수로캡슐화합니다. 10.2 절에서는전체모듈대신경량 Chisel 함수로작은빌딩블록을추상화하는방법을보여줍니다. 이러한 Chisel 함수는하드웨어인스턴스를생성하는데, 예를들어함수 sync 는입력에연결되고서로연결된두개의플립플롭을생성합니다. 이함수는두번째플립플롭의출력을반환합니다. 7.1 목록은모든입력처리회로를함수로보여줍니다. 유용하다면, 이러한함수는어떤유틸리티클래스객체로격상될수있습니다.

7.5 리셋동기화하기

모든디지털회로는레지스터를정의된상태로리셋하기위한리셋신호가필요합니다. Chisel 에서리셋상태는 RegInit 생성자로설정됩니다. 리셋신호는보통회로에대한비동기입력입니다. 즉, 플립플롭의리셋에직접연결하면타이밍제약을위반할수있습니다. 동기식리셋의경우플립플롭의셋업및홀드시간을위반할수있습니다. 또한비동기리셋입력으로사용되더라도, 여전히클록에동기화되어야합니다. 구체적으로, 리셋신호의 해제 *(release)* 는클록에동기화되어야합니다. 비동기리셋의또다른실패사례는회로의서로다른부분이서로다른두클록사이클에리셋되어결과적으로불일치하게되는것입니다.

```
def sync(v: Bool) = RegNext(RegNext(v))

def rising(v: Bool) = v & !RegNext(v)

def tickGen() = {
  val reg = RegInit(0.U(log2Up(fac).W))
  val tick = reg === (fac-1).U
  reg := Mux(tick, 0.U, reg + 1.U)
  tick
}

def filter(v: Bool, t: Bool) = {
  val reg = RegInit(0.U(3.W))
  when (t) {
    reg := reg(1, 0) ## v
  }
  (reg(2) & reg(1)) | (reg(2) & reg(0)) | (reg(1) & reg(0))
}

val btnSync = sync(io.btnU)

val tick = tickGen()
val btnDeb = RegInit(false.B)
when (tick) {
  btnDeb := btnSync
}

val btnClean = filter(btnDeb, tick)
val risingEdge = rising(btnClean)

// Use the rising edge of the debounced
// and filtered button for the counter
val reg = RegInit(0.U(8.W))
when (risingEdge) {
  reg := reg + 1.U
}
```

Listing 7.1: 함수로입력처리요약하기.

이문제에대한해결책은다른비동기입력과완전히동일한방식으로두개의플립플롭으로리셋신호를동기화하는것입니다.

리셋신호와클록신호는보통 Chisel 설계에서숨겨져있습니다. 그러나이신호들에접근하고설정하는것이가능합니다. 각모듈은암묵적필드 reset 을가집니다. 해결책은외부리셋신호의동기화를수행하고그동기화된신호를포함된모듈의리셋입력에연결하는최상위모듈을두는것입니다.

```
class SyncReset extends Module {
  val io = IO(new Bundle() {
    val value = Output(UInt(1.W))
  })

  val syncReset = RegNext(RegNext(reset))
  val cnt = Module(new WhenCounter(5))
  cnt.reset := syncReset

  io.value := cnt.io.cnt
}
```

위예제에서 SyncReset 은카운터 (WhenCounter) 를포함하는최상위모듈입니다. 최상위모듈의리셋신호는 reset 이라고불리며입력동기화기 (RegNext(RegNext(reset))) 에연결됩니다. 이입력동기화기의출력 (syncReset) 은카운터의리셋 입력 (cnt.reset := syncReset) 에연결됩니다.

7.6 연습문제

입력버튼에의해증가되는카운터를만드십시오. FPGA 보드의 LED 로카운터값을 2 진수로표시하십시오. 먼저, 입력버튼이바운스되는문제가있는지관찰하십시오. 그런다음 (1) 입력동기화기, (2) 디바운싱회로, (3) 잡음을억제하기위한다수결투표회로, (4) 카운터의증가를트리거하기위한에지검출회로로구성된완전한입력처리체인을구축하여그문제를해결하십시오.

현대의버튼이항상바운스된다는보장은없으므로, 빠른연속으로버튼을수동으로누르고낮은샘플주파수를사용하여바운싱과스파이크를시뮬레이션할수있습니다. 예를들어샘플주파수로 1 초를선택하십시오. 즉, 입력클록이 100 MHz 로동작한다면이를 100,000,000 으로나눕니다. 안정적인누름상태에정착하기전에빠른연속으로여러번버튼바운싱을시뮬레이션하십시오. 1 Hz 로샘플링하는디바운싱회로없이, 그리고있는상태로회로를테스트하십시오. 다수결투표를사용하면, 안정적인카운터증가를위해 1 초에서 2 초사이동안눌러야합니다. 또

한버튼의해제도다수결투표됩니다. 따라서회로는해제가 1 2 초보다길때만이를인식합니다.

8 유한상태기계

유한상태기계 (FSM) 는디지털설계의기본빌딩블록입니다. FSM 은 상태집합과상태간의조건부 (가드) 상태전이로기술될수있습니다. FSM 은리셋시설정되는초기상태를가집니다. FSM 은동기식순차회로라고도불립니다.

FSM 의구현은세부분으로구성됩니다: (1) 현재상태를보관하는레지스터, (2) 현재상태와입력에의존하여다음상태를계산하는조합논리, (3) FSM 의출력을계산하는조합논리.

원칙적으로, 상태를저장하기위한레지스터나다른메모리소자를포함하는모든디지털회로는단일 FSM 으로기술될수있습니다. 그러나이는실용적이지않을수있습니다. 예를들어, 여러분의노트북을단일 FSM 으로기술해보십시오. 다음장에서는더작은 FSM 들을결합하여통신하는 FSM 으로만듦으로써더큰시스템을구축하는방법을설명합니다.

8.1 기본유한상태기계

8.1 그림은 FSM 의개략도를보여줍니다. 레지스터는현재 state 를담고있습니다. 다음상태로직은현재 state 와입력 (in) 으로부터다음상태값 (nextState) 을계산합니다. 다음클록틱에서 state 는 nextState 가됩니다. 출력로직은출력 (out) 을계산합니다. 출력이현재상태에만의존하므로, 이상태기계는 무어머신이라고불립니다.

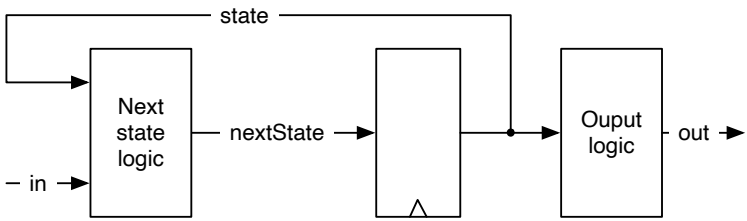


Figure 8.1: 유한상태기계 (무어타입).

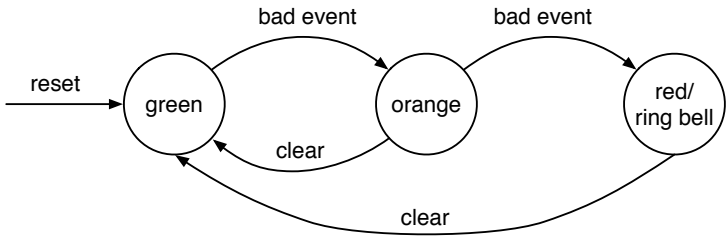


Figure 8.2: 경보 FSM 의상태다이어그램입니다.

상태다이어그램은이러한 FSM 의동작을시각적으로기술합니다. 상태다이어그램에서개별상태는상태이름으로레이블된원으로묘사됩니다. 상태전이는상태간의화살표로표시됩니다. 이전이가취해지는가드 (또는조건) 는화살표의레이블로그려집니다.

그림 8.2는간단한예제 FSM 의상태다이어그램을보여줍니다. 이 FSM 은경보수준을나타내는 녹색, 주황색, 빨간색세가지상태를가집니다. FSM 은 녹색수준에서시작합니다. 나쁜이벤트가발생하면경보수준이 주황색로전환됩니다. 두번째 bad event 가발생하면경보수준이 빨간색로전환됩니다. 이경우종을울리고자하며, 벨울리기이이 FSM 의유일한출력입니다. 이출력을 빨간색상태에추가합니다. 경보는 지우기신호로리셋할수있습니다.

상태다이어그램은시각적으로보기좋고 FSM 의기능을빠르게파악할수있게해주지만, 상태테이블이작성하기에는더빠를수있습니다. 표 8.1는우리의경보 FSM 에대한상태테이블을보여줍니다. 여기에는현재상태, 입력값, 그결과로나오는다음상태, 그리고현재상태에대한출력값이나열되어있습니다. 원칙적으로는가능한모든상태에대해가능한모든입력을명시해야합니다. 이표는 $3 \times 4 = 12$ 개의행을가지게될것입니다. 나쁜이벤트가발생할때는 지우기입력이 don't care 임을나타내어표를단순화합니다. 이는 나쁜이벤트가 지우기보다우선순위를가진다는의미입니다. 출력열에는일부반복이있습니다. 더큰 FSM 이나더많은출력이있다면, 표를다음상태논리를위한것과출력논리를위한것두개로나눌수있습니다.

마지막으로, 경보수준 FSM 의설계를마친후이를 Chisel 로코딩하겠습니다. 목록 8.1은경보 FSM 에대한 Chisel 코드를보여줍니다. FSM 의입력과출력에 Chisel 타입 Bool 을사용한다는점에유의하십시오. switch 제어명령을사용하려면 chisel3.util._ 를임포트해야합니다.

이간단한 FSM 의전체 Chisel 코드는한페이지에들어갑니다. 개별부분을하나씩살펴봅시다. FSM 에는두개의입력신호와하나의출력신호가있으며, 이는

```
import chisel3._
import chisel3.util._

class SimpleFsm extends Module {
  val io = IO(new Bundle{
    val badEvent = Input(Bool())
    val clear = Input(Bool())
    val ringBell = Output(Bool())
  })

  // The three states
  object State extends ChiselEnum {
    val green, orange, red = Value
  }
  import State._
  // The state register
  val stateReg = RegInit(green)

  // Next state logic
  switch (stateReg) {
    is (green) {
      when(io.badEvent) {
        stateReg := orange
      }
    }
    is (orange) {
      when(io.badEvent) {
        stateReg := red
      } .elsewhen(io.clear) {
        stateReg := green
      }
    }
    is (red) {
      when (io.clear) {
        stateReg := green
      }
    }
  }
  // Output logic
  io.ringBell := stateReg === red
}
```

Listing 8.1: 경보 FSM 에대한 Chisel 코드입니다.

Table 8.1: 경보 FSM 의상태테이블입니다.

	입력			
상태	나쁜이벤트	지우기	다음상태	벨울리기
초록	0	0	초록	0
초록	1	-	주황	0
주황	0	0	주황	0
주황	1	-	빨강	0
주황	0	1	초록	0
빨강	-	0	빨강	1
빨강	0	1	초록	1

Chisel Bundle 에담겨있습니다:

```
val io = IO(new Bundle{
  val badEvent = Input(Bool())
  val clear = Input(Bool())
  val ringBell = Output(Bool())
})
```

이부분에서최적의상태인코딩에대해논의할수도있을것입니다. 흔히사용되는두가지옵션은이진인코딩또는원-핫인코딩입니다. 그러나이러한저수준최적화는합성도구에맡기고, 가독성있는코드를목표로합니다.[1] 따라서상태에대한기호적이름을가진열거타입 ChiselEnum 을사용합니다:

```
object State extends ChiselEnum {
  val green, orange, red = Value
}
import State._
```

개별상태값은쉼표로구분된목록으로열거되며, 그뒤에 Value 대입이이어집니다. 상태를담고있는레지스터는 녹색상태를리셋값으로하여정의됩니다:

```
val stateReg = RegInit(green)
```

FSM 의핵심은다음상태논리에있습니다. 모든상태를다루기위해상태레지스터

[1] 현재버전의 Chisel 에서 ChiselEnum 타입은상태를이진인코딩으로나타냅니다. 원-핫인코딩등 다른인코딩을원한다면, 상태이름에대한 Chisel 상수를정의할수있습니다.

에대한 Chisel switch 를사용합니다. 각 is 분기안에서는, 입력에따라달라지는다음상태논리를, 상태레지스터에새로운값을대입함으로써코딩합니다:

```
switch (stateReg) {
  is (green) {
    when(io.badEvent) {
      stateReg := orange
    }
  }
  is (orange) {
    when(io.badEvent) {
      stateReg := red
    } .elsewhen(io.clear) {
      stateReg := green
    }
  }
  is (red) {
    when (io.clear) {
      stateReg := green
    }
  }
}
```

마지막으로, 상태가 빨간색일때참이되는 벨울림출력을코딩합니다.

```
io.ringBell := stateReg === red
```

Verilog 나 VHDL 에서흔히하듯이, 레지스터입력을위한 nextState 신호를도입하지 않았다는점에유의하십시오. Verilog 와 VHDL 의레지스터는특수한문법으로기술되며조합블록내에서대입 (및재대입) 될수없습니다. 따라서조합블록에서계산되는추가적인신호를도입하여레지스터입력에연결합니다. Chisel 에서는레지스터가기본타입이므로조합블록내에서자유롭게사용하고대입할수있습니다.

8.2 Mealy FSM 을이용한더빠른출력

Moore FSM 에서는출력이오직현재상태에만의존합니다. 이는입력의변화가출력의변화로보이는것이다음클록사이클보다빨리일어날수없다는것을의미합니다. 즉각적인변화를관찰하고자한다면, 입력에서출력으로이어지는조합경로가

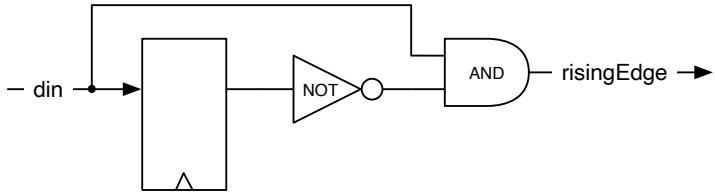


Figure 8.3: 상승에지감지기 (Mealy 타입 FSM) 입니다.

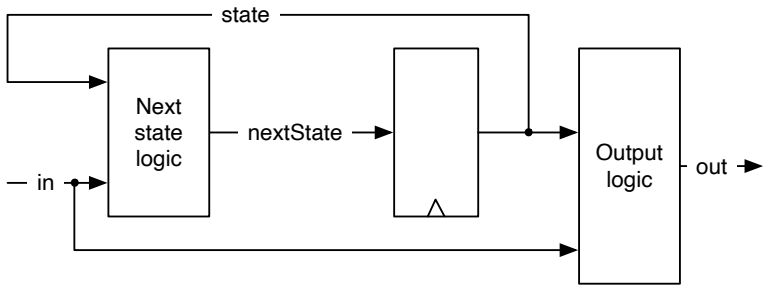


Figure 8.4: Mealy 타입유한상태기계입니다.

필요합니다. 최소한의예제인에지감지회로를살펴봅시다. 이 Chisel 한줄코드는이전에본적이있습니다:

```
val risingEdge = din & !RegNext(din)
```

그림 8.3는상승에지감지기의회로도를보여줍니다. 현재입력이 1 이고마지막클록사이클의입력이 0 이었을때, 출력은한클록사이클동안 1 이됩니다. 상태레지스터는단순히하나의 D 플립플롭이며, 다음상태는그저입력값입니다. 이를하나의클록사이클지연요소로도볼수있습니다. 출력논리는현재입력을현재상태와 비교합니다.

출력이입력에도의존하는경우, 즉 FSM 의입력과출력사이에조합경로가있는경우, 이를 Mealy 기계라고부릅니다.

그림 8.4는 Mealy 타입 FSM 의회로도를보여줍니다. Moore FSM 과유사하게, 레지스터는현재 state 를담고있으며, 다음상태논리는현재 state 와입력 (in) 으로부터다음상태값 (nextState) 을계산합니다. 다음클록틱에서 state 는 nextState 가됩니다. 출력논리는현재상태 그리고 FSM 에대한입력으로부터출

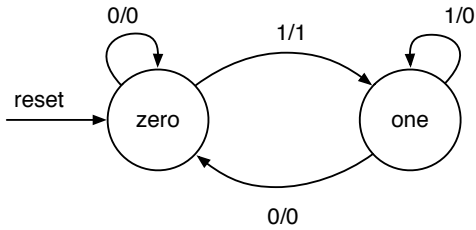


Figure 8.5: Mealy FSM 으로구현한상승에지검출기의상태다이어그램

력 (out) 을계산합니다.

그림 8.5는에지검출기를위한 Mealy FSM 의상태다이어그램을보여줍니다. 상태레지스터가단하나의 D 플립플롭으로만구성되어있으므로, 가능한상태는 두개뿐이며, 이예제에서는이를 zero 와 one 으로명명합니다. Mealy FSM 의 출력은상태뿐만아니라입력에도의존하므로, 출력을상태원안에표기하는방식으로는설명할수없습니다. 대신, 상태간의전이에는입력값 (조건) 과출력 (슬래시 뒤에) 모두가표시됩니다. 또한이제자기전이도그려야한다는점에유의하십시오. 예를들어, 상태 zero 에서입력이 0 이면 FSM 은상태 zero 에머무르며, 출력은 0 입니다. 상승에지 FSM 은상태 zero 에서상태 one 으로전이할때만 1 출력을 생성합니다. 입력이이제 1 임을나타내는상태 one 에서는출력이 0 입니다. 우리는입력의각상승에지마다단일 (사이클) 펄스만을원합니다.

목록 8.2은 Mealy 기계를이용한상승에지검출을위한 Chisel 코드를보여줍니다. 이전예제와마찬가지로, 단일비트입력과출력에는 Chisel 타입 Bool 을사용합니다. 이제출력논리는다음상태논리의일부가됩니다. zero 에서 one 으로전이할때출력은 true.B 로설정됩니다. 그렇지않으면출력에대한기본할당 (false.B) 이적용됩니다.

완전한형태의 FSM 이에지검출회로에최선의해법인지물을수있는데, 특히동일한기능을수행하는 Chisel 한줄짜리코드를이미살펴보았기때문입니다. 하드웨어소비량은비슷합니다. 두해법모두상태를위해단일 D 플립플롭이필요합니다. FSM 을위한조합논리는상태변화가현재상태와입력값에의존하기때문에아마조금더복잡할것입니다. 이기능의경우, 한줄짜리코드가작성하기도읽기도더쉬우며, 이점이더중요합니다. 따라서한줄짜리코드가선호되는해법입니다.

우리는가능한가장작은 Mealy FSM 중하나를보여주기위해이예제를사용했습니다. FSM 은세개이상의상태를가진더복잡한회로에사용되어야합니다.

```
import chisel3._
import chisel3.util._

class RisingFsm extends Module {
  val io = IO(new Bundle{
    val din = Input(Bool())
    val risingEdge = Output(Bool())
  })

  // The two states
  object State extends ChiselEnum {
    val zero, one = Value
  }
  import State._

  // The state register
  val stateReg = RegInit(zero)

  // default value for output
  io.risingEdge := false.B

  // Next state and output logic
  switch (stateReg) {
    is(zero) {
      when(io.din) {
        stateReg := one
        io.risingEdge := true.B
      }
    }
    is(one) {
      when(!io.din) {
        stateReg := zero
      }
    }
  }
}
```

Listing 8.2: Mealy FSM 을이용한상승에지검출

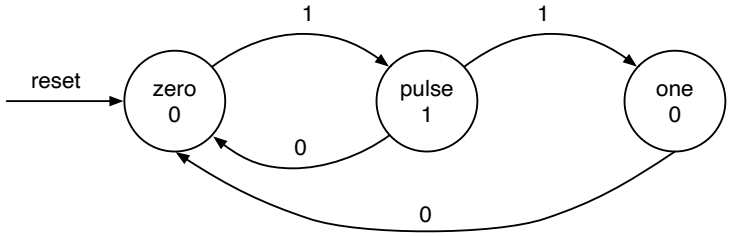


Figure 8.6: Moore FSM 으로구현한상승에지검출기의상태다이어그램

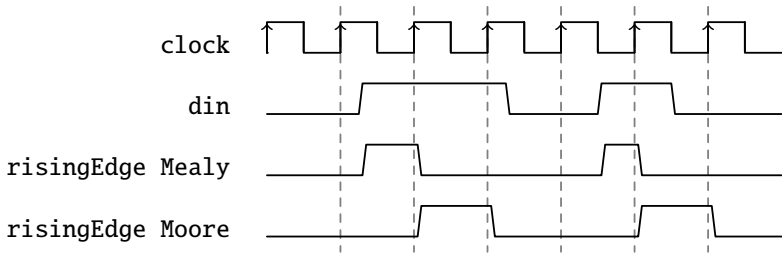


Figure 8.7: 상승에지검출을위한 Mealy FSM 과 Moore FSM 의파형

8.3 Moore 와 Mealy 비교

Moore FSM 과 Mealy FSM 의차이를보여주기위해, Moore FSM 으로에지검출을다시구현합니다.

그림 8.6는 Moore FSM 을이용한상승에지검출의상태다이어그램을보여줍니다. 가장먼저주목할점은 Moore FSM 은 Mealy 버전의두상태에비해세개의상태가필요하다는것입니다. 단일사이클펄스를생성하기위해서는상태 pulse 가필요합니다. FSM 은상태 pulse 에단한클록사이클만머무른후, 시작상태 zero 로되돌아가거나입력이다시 0 이되기를기다리는 one 상태로진행합니다. 입력조건은상태전이화살표위에, FSM 출력은상태를나타내는원안에표시합니다.

목록 8.3은상승에지검출회로의 Moore 버전을보여줍니다. 이는 Mealy 버전이나직접코딩한버전보다두배많은수의 D 플립플롭을사용합니다. 따라서결과적으로다음상태논리역시 Mealy 버전이나직접코딩한버전보다더큽니다.

그림 8.7는상승에지검출 FSM 의 Mealy 버전과 Moore 버전의파형을보여줍니다. Mealy 출력은입력의상승에지를밀접하게따라가는반면, Moore 출

```
import chisel3._
import chisel3.util._

class RisingMooreFsm extends Module {
  val io = IO(new Bundle{
    val din = Input(Bool())
    val risingEdge = Output(Bool())
  })

  // The three states
  object State extends ChiselEnum {
    val zero, puls, one = Value
  }
  import State._
  // The state register
  val stateReg = RegInit(zero)

  // Next state logic
  switch (stateReg) {
    is(zero) {
      when(io.din) {
        stateReg := puls
      }
    }
    is(puls) {
      when(io.din) {
        stateReg := one
      } .otherwise {
        stateReg := zero
      }
    }
    is(one) {
      when(!io.din) {
        stateReg := zero
      }
    }
  }

  // Output logic
  io.risingEdge := stateReg === puls
}
```

Listing 8.3: Moore FSM 을이용한상승에지검출

력은클록틱이후에상승함을확인할수있습니다. 또한 Moore 출력은한클록사이클의폭을가지는반면, Mealy 출력은대개한클록사이클보다짧다는것을확인할수있습니다.

위의예제로부터, Mealy FSM 이더적은상태 (따라서더적은논리) 를필요로하고 Moore FSM 보다더빠르게반응하므로 더나은 FSM 이라고여기고싶은유혹을받게됩니다. 그러나 Mealy 기계내부의조합경로는더큰설계에서문제를일으킬수있습니다. 첫째, 통신하는 FSM 들의연쇄에서 (다음장참조) 이조합경로는길어질수있습니다. 둘째, 통신하는 FSM 들이순환을이루면, 그결과는조합루프가되는데, 이는동기식설계에서오류입니다. Moore FSM 에서는상태레지스터로인해조합경로가끊어지므로, 통신하는 Moore FSM 에서는위의문제들이전혀발생하지않습니다.

요약하자면, Moore FSM 은통신하는상태기계들을결합하는데더유리하며, Mealy FSM 보다 더견고합니다. Mealy FSM 은동일한클록사이클내의반응이무엇보다중요할때만사용하십시오. 상승에지검출과같이사실상 Mealy 기계인작은회로들도물론괜찮습니다.

8.4 연습문제

이장에서는매우작은 FSM 의여러예제를살펴보았습니다. 이제 실제 FSM 코드를작성할차례입니다. 조금더복잡한예제를선택하여 FSM 을구현하고이를위한테스트벤치를작성하십시오.

FSM 의대표적인예로신호등제어기가있습니다 (참고: [6, Section 14.3]). 신호등제어기는빨간불에서초록불로전환될때, 교차로의두도로가모두진행금지신호 (빨간불과주황불) 인중간단계를거치도록보장해야합니다. 이예제를조금더흥미롭게만들기위해우선도로를고려해봅시다. 부도로에는 (교차로로진입하는두지점모두에) 차량감지기가두개있습니다. 차량이감지될때만부도로를초록불로전환하고, 그다음다시우선도로를초록불로전환하십시오.

TODO: Luca: 유클리드알고리즘을이용한최대공약수도좋은연습문제가될수있습니다. *Martin:* 하지만이는 *Chisel* 홈페이지에 *FSM* 없이소개되어있습니다.

TODO: 여기더흥미로운연습문제가있습니다. 그리고이는 *Dally* 에서가져온것이아닙니다.

9 통신하는상태기계

문제는종종단일 FSM 만으로기술하기에는너무복잡합니다. 이경우, 문제를두개이상의더작고단순한 FSM 으로나눌수있습니다. 이렇게나뉜 FSM 들은신호를통해서로통신합니다. 한 FSM 의출력이다른 FSM 의입력이되며, 한 FSM 이다른 FSM 의출력을관찰합니다. 큰 FSM 을더단순한 FSM 들로나누는것을 FSM 인수분해라고부릅니다. 통신하는 FSM 들은흔히설계사양으로부터직접설계됩니다. 애초에해당설계를위한단일 FSM 은너무커질것입니다.

9.1 라이트플래셔예제

통신하는 FSM 을다루기위해, [6, Chapter 17] 의예제인라이트플래셔를사용합니다. 라이트플래셔는입력 start 하나와출력 light 하나를가집니다. 라이트플래셔의명세는다음과같습니다.

- start 가한클록사이클동안하이일때깜박임시퀀스가시작됩니다.
- 이시퀀스는세번깜박이는것입니다.
- 이때 light 는여섯클록사이클동안 켜지고, 깜빡임사이에는 light 가네클록사이클동안 꺼집니다;
- 시퀀스가끝난후, FSM 은 light 를 끄고다음시작을기다립니다.

직접구현한 FSM[1]은 27 개의상태를가집니다. 즉, 입력을기다리는초기상태하나, 세번의 켜짐상태를위한 3×6 개의상태, 그리고 꺼짐상태를위한 2×4 개의상태입니다. 이렇게단순한방식으로구현한라이트플래셔의코드는보이지않겠습니다.

이문제는이큰 FSM 을더작은두개의 FSM 으로분해함으로써더우아하게해결할수있습니다. 마스터 FSM 은깜박임로직을구현하고, 타이머 FSM 은대기를구현합니다. 그림 9.1는두 FSM 의구성을보여줍니다.

타이머 FSM 은원하는타이밍을만들어내기위해 6 또는 4 클록사이클동안카운트다운합니다. 타이머의명세는다음과같습니다.

[1] 상태다이어그램은 [6, p. 376] 에나와있습니다.

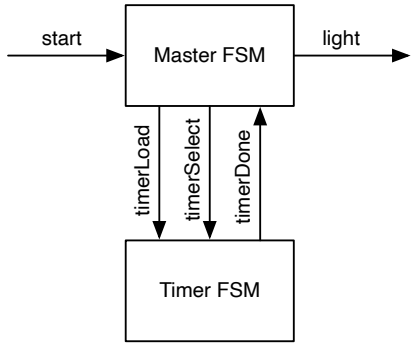


Figure 9.1: 마스터 FSM 과타이머 FSM 으로분할된라이트플래셔입니다.

- timerLoad 가어서트되면, 타이머는상태와무관하게다운카운터에값을로드합니다.
- timerSelect 는로드할값으로 5 와 3 중하나를선택합니다.
- timerDone 은카운터가카운트다운을완료하면어서트되고, 계속어서트된상태를유지합니다.
- 그렇지않으면, 타이머는카운트다운을진행합니다.

다음코드는라이트플래셔의타이머 FSM 을보여줍니다.

```
val timerReg = RegInit(0.U)
timerDone := timerReg === 0.U

// Timer FSM (down counter)
when(!timerDone) {
  timerReg := timerReg - 1.U
}
when (timerLoad) {
  when (timerSelect) {
    timerReg := 5.U
  } .otherwise {
    timerReg := 3.U
  }
}
```

Listing 9.1는마스터 FSM 을보여줍니다. 이는시작상태 off 와전체깜박임시퀀스를위한상태들을가집니다. 각상태에서는시간이완료되기를기다립니다. 타이머는완료될때마다, 그리고초기 off 상태에서로드됩니다. 신호 timerSelect 는다음상태의다운카운터값을선택합니다.

```
object State extends ChiselEnum {
  val off, flash1, space1, flash2, space2, flash3 = Value
}
import State._

val stateReg = RegInit(off)

val light = WireDefault(false.B) // FSM output

// Timer connection
val timerLoad = WireDefault(false.B) // start timer
val timerSelect = WireDefault(true.B) // 6 or 4 cycles
val timerDone = Wire(Bool())

timerLoad := timerDone

// Master FSM
switch(stateReg) {
  is(off) {
    timerLoad := true.B
    timerSelect := true.B
    when (start) { stateReg := flash1 }
  }
  is (flash1) {
    timerSelect := false.B
    light := true.B
    when (timerDone) { stateReg := space1 }
  }
  is (space1) {
    when (timerDone) { stateReg := flash2 }
  }
  is (flash2) {
    timerSelect := false.B
    light := true.B
    when (timerDone) { stateReg := space2 }
  }
```

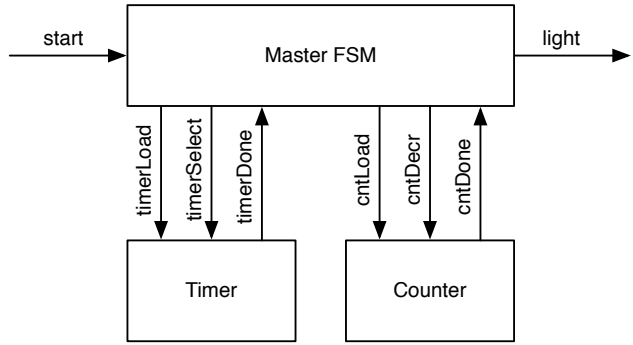


Figure 9.2: 마스터 FSM, 타이머 FSM, 카운터 FSM 으로분할된라이트플래셔입니다.

```
    is (space2) {
      when (timerDone) { stateReg := flash3 }
    }
    is (flash3) {
      timerSelect := false.B
      light := true.B
      when (timerDone) { stateReg := off }
    }
  }
```

Listing 9.1: 라이트플래셔의마스터 FSM 입니다.

마스터 FSM 과타이머로이루어진이해법에도마스터 FSM 의코드에는여전히중복이있습니다. 상태 flash1, flash2, flash3 은동일한기능을수행하며, 상태 space1 과 space2 도마찬가지입니다. 남은깜박임횟수를두번째카운터로분리해낼수있습니다. 그러면마스터 FSM 은 off, flash, space 세상태로줄어듭니다.

그림 9.2는마스터 FSM 과카운트를수행하는두개의 FSM 으로이루어진설계를보여줍니다. 하나의 FSM 은 켜짐과 꺼짐의구간길이를위한클록사이클을 카운트하고, 두번째 FSM 은남은깜박임횟수를카운트합니다. Listing 9.2 코드는다운카운터 FSM 을보여줍니다.

```
val cntReg = RegInit(0.U)
cntDone := cntReg === 0.U
```

```
// Down counter FSM
when(cntLoad) { cntReg := 2.U }
when(cntDecr) { cntReg := cntReg - 1.U }
```

Listing 9.2: 다운카운터 FSM 입니다.

카운터는 3 번의깜박임에대해 2 로로드된다는점에유의하십시오. 이는 남은깜박임횟수를카운트하는것이며, 타이머가완료되었을때상태 space 에서감소되기때문입니다. Listing 9.3는이중으로리팩터링된플래셔의마스터 FSM 을보여줍니다.

```
object State extends ChiselEnum {
  val off, flash, space = Value
}
import State._

val stateReg = RegInit(off)

val light = WireDefault(false.B) // FSM output

// Timer connection
val timerLoad = WireDefault(false.B) // start timer with a
    load
val timerSelect = WireDefault(true.B) // select 6 or 4
    cycles
val timerDone = Wire(Bool())
// Counter connection
val cntLoad = WireDefault(false.B)
val cntDecr = WireDefault(false.B)
val cntDone = Wire(Bool())

timerLoad := timerDone

switch(stateReg) {
  is(off) {
    timerLoad := true.B
    timerSelect := true.B
    cntLoad := true.B
    when (start) { stateReg := flash }
  }
  is (flash) {
```

```
      timerSelect := false.B
      light := true.B
      when (timerDone & !cntDone) { stateReg := space }
      when (timerDone & cntDone) { stateReg := off }
    }
    is (space) {
      cntDecr := timerDone
      when (timerDone) { stateReg := flash }
    }
  }
```

Listing 9.3: 이중으로리팩터링된라이트플래셔의마스터 FSM 입니다.

마스터 FSM 이단세개의상태로축소되었을뿐만아니라, 현재솔루션은설정변경도더용이합니다. 켜짐또는 꺼짐구간의길이나깜빡임횟수를변경하고자할때 어떤 FSM 도변경할필요가없습니다.

이절에서는제어신호만을주고받는통신회로, 특히 FSM 을살펴보았습니다. 계산을수행하려면다음절에서다루는것처럼 FSM 을데이터패스와결합할수있습니다.

9.2 데이터패스가있는상태기계

통신하는상태기계의대표적인예중하나는데이터패스와결합된상태기계입니다. 이러한결합은흔히데이터패스를갖는유한상태기계 (FSMD) 라고불립니다. 상태기계는데이터패스를제어하고, 데이터패스는계산을수행합니다. FSM 의입력은환경으로부터의입력과데이터패스로부터의출력입니다. 환경으로부터의일부데이터역시데이터패스로입력되며, 데이터출력은데이터패스로부터나옵니다.

그림 9.3에나타난 FSMD 는 popcount, 즉 해밍무게라고도불리는값을계산하는예시로사용됩니다. 해밍무게란영기호와다른기호의개수입니다. 이진문자열의경우이는 '1' 의개수입니다.

popcount 유닛은데이터입력 din 과결과출력 popCount 를포함하며, 둘다데이터패스에연결됩니다. 입력과출력에는 ready/valid 핸드셰이크를사용합니다. 데이터가사용가능할때 valid 가어서트됩니다. 수신자가데이터를받을수있을때 ready 를어서트합니다. 두신호가모두어서트되면전송이이루어집니다. 핸드셰이크신호는 FSM 에연결됩니다. FSM 은제어신호를통해데이터패스와, 상태신호를통해데이터패스로부터연결됩니다.

우리는 FSM 과데이터패스를함께설계할것입니다. 그림 9.4는 FSM 의상태다이어그램을보여주고, 그림 9.5는 popcount 회로의데이터패스를보여줍

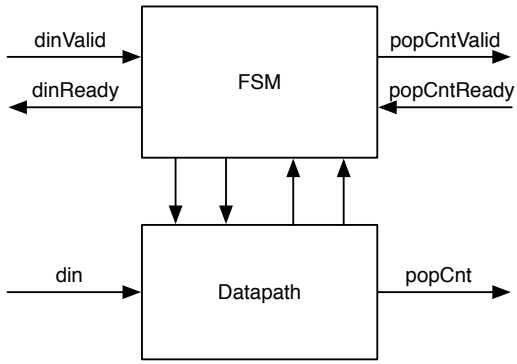


Figure 9.3: 데이터패스가있는상태기계.

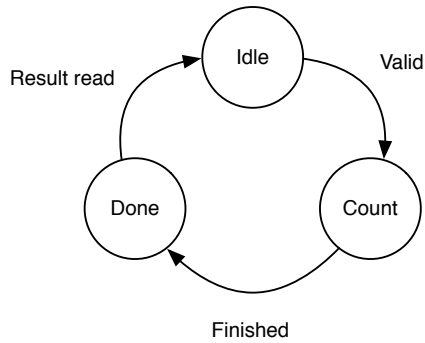


Figure 9.4: popcount FSM 의상태다이어그램.

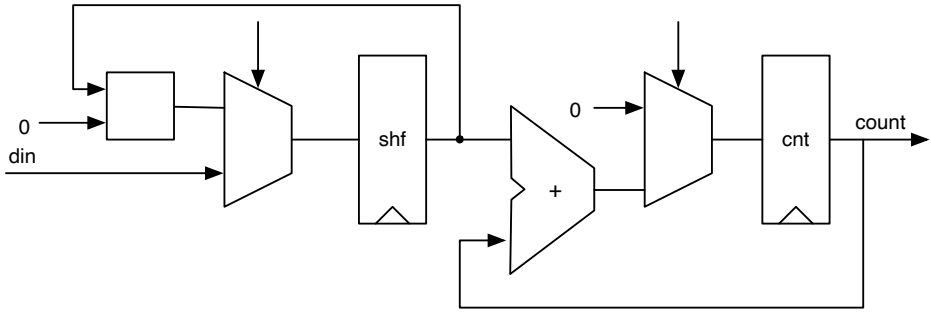


Figure 9.5: popcount 회로의데이터패스.

니다. FSM 은상태 Idle 에서시작하며, 이상태에서 FSM 은입력을기다립니다. dinValid 가어서트되어신호되는것처럼데이터가도착하면 FSM 은시프트레지스터를로드하고상태 Count 로진행합니다. 데이터는 shf 레지스터에로드됩니다. 로드시 cnt 레지스터도 0 으로리셋됩니다.

상태 Count 에서는 '1' 의개수가순차적으로카운트됩니다. 이계산을수행하기위해시프트레지스터, 가산기, 누산레지스터, 그리고 (데이터패스에는나타나지않는) 다운카운터를사용합니다. '1' 의개수를세기위해 shf 레지스터를오른쪽으로시프트하고, 매클록사이클마다최하위비트를 cnt 에더합니다. 그림에는나타나지않는카운터가모든비트가최하위비트를통해시프트될때까지카운트다운합니다. 카운터가 0 에도달하면 popcount 가완료된것입니다. FSM 은상태 Done 으로전환하고 popCntReady 를어서트하여결과를신호합니다. popCntValid 가어서트되어신호되는것처럼결과가읽히면 FSM 은다시 Idle 로전환되어다음 popcount 를계산할준비를합니다.

목록 9.4에나타난최상위컴포넌트는 FSM 과데이터패스컴포넌트를인스턴스화하고이들을연결합니다. 목록 9.5는 popcount 회로의데이터패스에대한 Chisel 코드를보여줍니다. load 신호가있을때 regData 레지스터는입력으로로드되고, regPopCount 레지스터는 0 으로리셋되며, 카운터레지스터 regCount 는수행할시프트횟수로설정됩니다. 그렇지않으면 regData 레지스터가오른쪽으로시프트되고, regData 레지스터의최하위비트가 regPopCount 레지스터에더해지며, 카운터는 0 이될때까지감소합니다. 카운터가 0 이되면출력에는 popcount 가담깁니다.

```
class PopCountDataPath extends Module {
  val io = IO(new Bundle {
    val din = Input(UInt(8.W))
```

```
class PopulationCount extends Module {
  val io = IO(new Bundle {
    val dinValid = Input(Bool())
    val dinReady = Output(Bool())
    val din = Input(UInt(8.W))
    val popCntValid = Output(Bool())
    val popCntReady = Input(Bool())
    val popCnt = Output(UInt(4.W))
  })


  val fsm = Module(new PopCountFSM)
  val data = Module(new PopCountDataPath)

  fsm.io.dinValid := io.dinValid
  io.dinReady := fsm.io.dinReady
  io.popCntValid := fsm.io.popCntValid
  fsm.io.popCntReady := io.popCntReady

  data.io.din := io.din
  io.popCnt := data.io.popCnt
  data.io.load := fsm.io.load
  fsm.io.done := data.io.done
}
```

Listing 9.4: popcount 회로의최상위레벨.

```
    val load = Input(Bool())
    val popCnt = Output(UInt(4.W))
    val done = Output(Bool())
  })

  val dataReg = RegInit(0.U(8.W))
  val popCntReg = RegInit(0.U(8.W))
  val counterReg= RegInit(0.U(4.W))

  dataReg := 0.U ## dataReg(7, 1)
  popCntReg := popCntReg + dataReg(0)

  val done = counterReg === 0.U
  when (!done) {
    counterReg := counterReg - 1.U
  }

  when(io.load) {
    dataReg := io.din
    popCntReg := 0.U
    counterReg := 8.U
  }

  // debug output
  printf("%x %d\n", dataReg, popCntReg)

  io.popCnt := popCntReg
  io.done := done
}
```

Listing 9.5: popcount 회로의데이터패스.

```
class PopCountFSM extends Module {
  val io = IO(new Bundle {
    val dinValid = Input(Bool())
    val dinReady = Output(Bool())
    val popCntValid = Output(Bool())
    val popCntReady = Input(Bool())
    val load = Output(Bool())
    val done = Input(Bool())
  })

  object State extends ChiselEnum {
    val idle, count, done = Value
  }
  import State._
  val stateReg = RegInit(idle)
  io.load := false.B
  io.dinReady := false.B
  io.popCntValid := false.B

  switch(stateReg) {
    is(idle) {
      io.dinReady := true.B
      when(io.dinValid) {
        io.load := true.B
        stateReg := count
      }
    }
    is(count) {
      when(io.done) {
        stateReg := done
      }
    }
    is(done) {
      io.popCntValid := true.B
      when(io.popCntReady) {
        stateReg := idle
      }
    } } }
```

Listing 9.6: popcount 회로의 FSM.

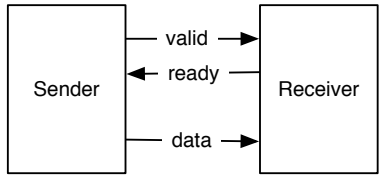


Figure 9.6: ready/valid 흐름제어.

목록 9.6은 FSM 의코드를보여줍니다. FSM 은상태 idle 에서시작합니다. 입력데이터에대한 valid 신호 (dinValid) 가있으면 count 상태로전환되어데이터 패스가카운트를마칠때까지기다립니다. popcount 가 done 이되면 FSM 은상태 done 으로전환되어 popcount 가읽힐때 (즉 popCntReady 로신호될때) 까지기다립니다.

popcount 예시는데이터 (워드) 를소비하고데이터 (popcount) 를생성했습니다. 데이터의조율된교환을위해핸드셰이크신호를사용합니다. 다음절에서는단방향데이터교환의흐름제어를위한 ready/valid 인터페이스를설명합니다.

9.3 Ready/Valid 인터페이스

서브시스템간의통신은데이터의이동과흐름제어를위한핸드셰이킹으로일반화될 수있습니다. popcount 예시에서우리는 valid 와 ready 신호를사용하는입력 및출력데이터에대한핸드셰이킹인터페이스를살펴보았습니다.

ready/valid 인터페이스 [6, p. 480] 는송신측 (생산자또는소스라고도불림) 의 data 와 valid 신호, 그리고수신측 (소비자또는목적지라고도불림) 의 ready 신호로구성되는단순한흐름제어인터페이스입니다. 그림 9.6는 ready/valid 연결을보여줍니다. 송신자는 data 가사용가능할때 valid 를어서트하고, 수신자는한워드의데이터를받을준비가되었을때 ready 를어서트합니다. 데이터전송은 valid 와 ready 두신호가모두어서트될때발생합니다. 두신호중하나라도어서트되지않으면전송은이루어지지않습니다.

그림 9.7은송신자에게데이터가있기전에 (클록사이클 2 부터) 수신자가 ready 를신호하는 ready/valid 트랜잭션의타이밍다이어그램을보여줍니다. 데이터전송은클록사이클 4 에서발생합니다. 클록사이클 5 부터는송신자에게데이터가없고, 수신자도다음전송에대해준비되어있지않습니다. 수신자가매클록사이클마다데이터를받을수있는경우, 이를” 항상 ready” 인터페이스라부르며 ready 는 true 로하드코딩될수있습니다.

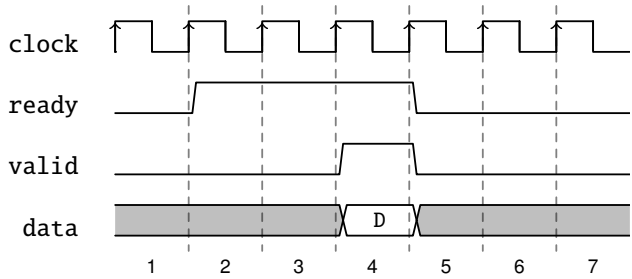


Figure 9.7: ready/valid 인터페이스를이용한데이터전송, 이른 ready.

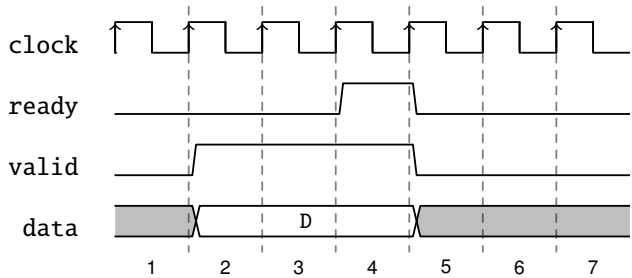


Figure 9.8: ready/valid 인터페이스를이용한데이터전송, 늦은 ready.

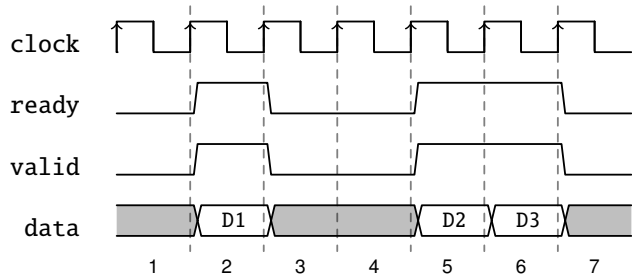


Figure 9.9: 단일사이클 ready/valid 및연속전송.

그림 9.8는수신자가준비되기전에송신자가 valid 를신호 (클록사이클 2 부터) 하는 ready/valid 트랜잭션의타이밍다이어그램을보여줍니다. 데이터전송은클록사이클 4 에서일어납니다. 클록사이클 5 부터는송신자에게데이터가없고수신자도다음전송을준비하지않은상태입니다. " 항상준비됨" 인터페이스와유사하게, 항상유효한인터페이스도생각해볼수있습니다. 다만이경우데이터는 ready 신호에따라변경되지않을가능성이높으며, 그러면우리는핸드셰이크신호를그냥생략하게될것입니다.

그림 9.9는 ready/valid 인터페이스사용의추가변형을보여줍니다. 클록사이클 2 에서는두신호 (ready 와 valid) 가단한클록사이클동안만어서트되는일이발생하며, 이때 D1 의데이터전송이일어납니다. 데이터는클록사이클 5 와 6 에서 D2 와 D3 의전송에서보듯이 (매클록사이클마다) 연속으로전송될수있습니다.

이인터페이스를조합가능하게만들기위해서는, ready 와 valid 어느쪽도다른신호에조합적으로의존해서는안됩니다. 이인터페이스가매우흔하기때문에, Chisel 은다음과유사한 DecoupledIO 번들을정의합니다:

```
class DecoupledIO[T <: Data](gen: T) extends Bundle {
  val ready = Input(Bool())
  val valid = Output(Bool())
  val bits  = Output(gen)
}
```

DecoupledIO 번들은 data 의타입으로매개변수화됩니다. Chisel 이정의한이인터페이스는데이터에 bits 필드를사용합니다. DecoupledIO 는 chisel3.util 패키지의일부입니다.

ready 또는 valid 가어서트된후데이터전송이 일어나지않은상태에서다시디어

서트될수있는지는여전히의문으로남습니다. 예를들어, 수신자가한동안준비된상태였다가데이터를받지못했지만, 다른이벤트로인해준비되지않은상태로바뀔수있습니다. 송신자의경우도마찬가지로, 몇클록사이클동안만데이터가유효한상태였다가데이터전송이일어나지않은채로유효하지않은상태가되는것을생각해볼수있습니다. 이러한동작이허용되는지여부는 ready/valid 인터페이스자체의일부가아니며, 인터페이스의구체적인사용방식에따라정의되어야합니다.

Chisel 은 DecoupledIO 클래스를사용할때 ready 와 valid 의신호방식에아무런요구사항을두지않습니다. 하지만 IrrevocableIO 클래스는송신자에게다음과같은제약을둡니다:

> valid 가하이이고 ready 가로우인사이클이후 bits 의값을변경하지않을것을약속하는 ReadyValidIO 의구체적인서브클래스입니다. 또한 valid 가한번올라가면 ready 도올라간이후가아니고서는결코내려가지않습니다.

이것은 단지관례일뿐이며, IrrevocableIO 클래스를사용하는것만으로는강제될수없다는점에유의하십시오.

AXI 버스 [3] 는버스의다음각부분에대해하나씩의 ready/valid 인터페이스를사용합니다: 읽기주소, 읽기데이터, 쓰기주소, 쓰기데이터. AXI 는송신자가일단 valid 를어서트하면데이터전송이일어날때까지이를디어서트할수없도록인터페이스를제한합니다. 이는앞서 IrrevocableIO 인터페이스의설명에서기술한것과동일한제약입니다. 또한송신자는 valid 를어서트하기위해수신자의 ready 를기다려서는안됩니다. 수신자측은좀더완화되어있습니다. ready 가어서트된경우, valid 가어서트되기전에이를디어서트하는것이허용됩니다. 또한수신자는 ready 를어서트하기전에 valid 가어서트되기를기다릴수있습니다.

목록 9.7는 ready/valid 인터페이스사용의예를보여줍니다. 이회로는레지스터로구성된버퍼를나타냅니다. 이버퍼는입력에 ready/valid 인터페이스 (DecoupledIO) 를하나, 출력에도하나를갖고있습니다. DecoupledIO 번들은송신자의관점에서정의됩니다. 따라서버퍼의입력 (in) 은 Flipped 로방향을바꿔야합니다.

이모듈은데이터용레지스터 (dataReg) 와, 버퍼가비어있는지가득찼는지를신호하는 1 비트레지스터 (emptyReg) 를포함합니다. 이단일비트는 empty 와 full 이라는두상태를갖는 2 상태무어 FSM 을나타냅니다. 입력 ready 신호와출력 valid 신호는오직 emptyReg 의상태에만의존합니다. 버퍼의입력과출력사이에는조합경로가존재하지않습니다.

버퍼가비어있고입력에유효한데이터가있을때, 데이터는레지스터에저장되고상태는 full 로바뀝니다. 버퍼가가득차있고소비자측에서준비되었다는신호를보낼때, 데이터는읽힌것으로간주되고버퍼는다시비게됩니다.

```
class ReadyValidBuffer extends Module {
  val io = IO(new Bundle{
    val in = Flipped(new DecoupledIO(UInt(8.W)))
    val out = new DecoupledIO(UInt(8.W))
  })

  val dataReg = Reg(UInt(8.W))
  val emptyReg = RegInit(true.B)

  io.in.ready := emptyReg
  io.out.valid := !emptyReg
  io.out.bits := dataReg

  when (emptyReg & io.in.valid) {
    dataReg := io.in.bits
    emptyReg := false.B
  }

  when (!emptyReg & io.out.ready) {
    emptyReg := true.B
  }
}
```

Listing 9.7: ready/valid 인터페이스를갖춘버퍼로서의레지스터

10 하드웨어생성기

Chisel 의강점은하드웨어생성기를작성할수있게해준다는점입니다. VHDL 이나 Verilog 와같은이전세대의하드웨어기술언어에서는대개 Java 나 Python 같은또다른언어를사용해하드웨어를생성했습니다. Chisel 을사용하기전에, 저는종종 VHDL 테이블을생성하기위한작은 Java 프로그램을작성했습니다. Chisel 에서는하드웨어구성시점에 Scala 와 Java 의강력한기능전체와그다양한오픈소스라이브러리를사용할수있습니다. 따라서우리는하드웨어생성기를같은언어로작성하여 Chisel 회로생성과정의일부로실행할수있습니다. 추가적인 생성기예제는 [24] 에서찾아볼수있습니다.

10.1 Scala 맛보기

이절에서는 Scala 에대한아주간략한소개를제공합니다. Chisel 용간단한하드웨어생성기를작성하는데는이정도면충분할것입니다. Scala 에대한심도있는소개를원하신다면, Odersky 등의교재 [29] 를추천합니다. Scala 웹사이트에는 온라인 Scala 책도실려있습니다.[1]

Scala 에는두종류의변수, 즉 val 과 var 가있습니다. val 은표현식에이름을부여하며다른값으로재할당할수없습니다. 다음코드조각은 zero 라는이름의정수값을정의하는예를보여줍니다. zero 에다른값을재할당하려고하면컴파일오류가 발생합니다.

```
// A value is a constant
val zero = 0
// No new assignment is possible
// The following will not compile
zero = 3
```

Chisel 에서는 val 을사용해하드웨어컴포넌트에이름을붙입니다. := 연산자는 Chisel 연산자이지 Scala 연산자가아니라는점에유의하십시오.

[1] Chisel 이아직 Scala 2 를기반으로하기때문에, 이링크는해당책의 Scala 2 버전을가리킵니다.

Scala 는또한 var 로서좀더전통적인형태의가변변수도제공합니다. 다음코드는정수변수를정의하고새로운값을재할당합니다:

```
// We can change the value of a var variable
var x = 2
x = 3
```

하드웨어 생성기를작성하려면 Scala var 가필요하지만, 하드웨어 컴포넌트의이름을붙이는데는 var 가전혀필요하지않습니다.

그변수들이어떤타입을가지는지궁금하셨을수있습니다. 위예제에서정수상수를할당했으므로변수타입은 추론되며, 이는 Scala Int 타입입니다. 대부분의경우 Scala 컴파일러가타입을추론할수있습니다. 하지만좀더명시적으로표현하고싶다면다음과같이타입을명시적으로지정할수있습니다.

```
val number: Int = 42
```

간단한루프는다음과같이작성합니다.

```
// Loops from 0 to 9
// Automatically creates loop value i
for (i <- 0 until 10) {
  println(i)
}
```

회로생성기에는루프를사용합니다. 다음루프는시프트레지스터의개별비트를연결합니다.

```
val regVec = Reg(Vec(8, UInt(1.W)))

regVec(0) := io.din
for (i <- 1 until 8) {
  regVec(i) := regVec(i-1)
}
```

이것이시프트레지스터를가장간결하게표현한방식은아니라는점에유의하십시오. 적절한크기의일반 UInt 를사용하고, ## 연산자와적절한인덱싱을이용한표현식으로레지스터의새값을할당하는편이더낫습니다. 이코드스니펫은 Scala for 루프를회로생성에어떻게사용할수있는지보여줍니다.

조건은 if 와 else 로표현합니다. 이조건은회로생성중 Scala 런타임에서평가된다는점에유의하십시오. 이구문은멀티플렉서를만들어내는것이 아니라, 설정가능한하드웨어생성기를작성할수있게해줍니다.

```
for (i <- 0 until 10) {
  print(i)
  if (i%2 == 0) {
    println(" is even")
  } else {
    println(" is odd")
  }
}
```

Scala 에는 튜플이라는개념이있습니다. 튜플은서로다른타입들의시퀀스를 담을수있습니다. 튜플은개별필드들을괄호안에배치하여구성합니다. 필드에는 ._n 으로접근하며, 첫번째필드는 1 부터시작합니다. 다음코드는우편번호와이 름을가진도시를나타내는튜플을생성합니다.

```
val city = (2000, "Frederiksberg")
val zipCode = city._1
val name = city._2
```

튜플은함수에서둘이상의값을반환하고자할때유용합니다. 튜플을이용하면출력 이둘이상인 Chisel 컴포넌트를, 본격적인모듈대신경량함수로표현할수있습니 다.

Scala 에는강력한 컬렉션라이브러리가있습니다. 그중더간단한컬렉션타입 중하나가 Seq 로, 원소들의순서가있는컬렉션 (시퀀스라고도함) 입니다. 기본구 현은불변입니다. Seq 에는 () 로인덱싱하며, 인덱싱은 0 부터시작합니다. Seq 는여러서로다른구현을가진베이스클래스입니다. 하지만대부분의 Chisel 하드 웨어생성기에서는 Seq 를직접사용하는것이선호되는방식입니다. 다음코드는 Scala Int 값네개를담는 Seq 를생성하는방법을보여줍니다. 두번째줄은두번째 원소에접근하며, second 는 15 가됩니다.

```
val numbers = Seq(1, 15, -2, 0)
val second = numbers(1)
```

10.2 함수를이용한경량컴포넌트

모듈은하드웨어기술을구조화하는표준적인방법입니다. 하지만모듈을선언하고 인스턴스화하여연결할때는일종의보일러플레이트코드가발생합니다. 하드웨어 를구조화하는경량방법으로는함수를사용하는방법이있습니다. Scala 함수는 Chisel(및 Scala) 매개변수를받아생성된하드웨어를반환할수있습니다. 간단

한예로, 가산기를생성해보겠습니다.

```
def adder (x: UInt, y: UInt) = {
  x + y
}
```

Scala 에서함수의반환값은마지막표현식의결과입니다.[2] 이제 adder 함수를호출하기만하면가산기두개를생성할수있습니다.

```
val x = adder(a, b)
// another adder
val y = adder(c, d)
```

이것이 하드웨어생성기라는점에유의하십시오. 이코드는엘라보레이션중에어떤덧셈연산도실행하지않으며, 대신가산기두개 (하드웨어인스턴스) 를생성합니다. 다시말해, 이코드는가산기출력으로향하는와이어를반환합니다. 우리는첫번째하드웨어생성기를작성한것입니다!

가산기는단순함을위해만든인위적인예제입니다. Chisel 에는이미 +(that: UInt) 와같은가산기생성기함수가있습니다.

경량하드웨어생성기로서함수는 (레지스터를이용해) 상태를담을수도있습니다. 다음예제는한클록사이클지연요소 (레지스터) 를반환합니다. 함수가단일문장으로이루어져있다면, 한줄로작성하고중괄호 ({}) 를생략할수있습니다.

```
def delay(x: UInt) = RegNext(x)
```

함수자체를매개변수로하여함수를호출함으로써, 이는두클록사이클의지연을생성했습니다.

```
val delOut = delay(delay(delIn))
```

다시말하지만, RegNext() 가이미지연을위한레지스터를생성하는그함수이므로, 이는유용하기에는너무짧은예제입니다.

함수는값을하나만반환합니다. 하나이상의출력을제공하려면, 여러출력와이어를 Scala 튜플로감쌀수있습니다. 다음코드는두입력을비교하고두출력을갖는하드웨어를생성합니다.

```
def compare(a: UInt, b: UInt) = {
  val equ = a === b
  val gt = a > b
```

[2] Scala 에는 return 문도있습니다. 이코드는 return x + y 처럼조금더장황하게작성할수도있었습니다.

```
    (equ, gt)
  }
```

괄호를사용하여, 비교기회로의출력에연결된두와이어를 Scala 튜플로감쌉니다.

compare 함수로비교기컴포넌트를생성하면두와이어의튜플이반환됩니다. ._n 문법으로두와이어에접근할수있습니다.

```
val cmp = compare(inA, inB)
val equResult = cmp._1
val gtResult = cmp._2
```

하지만다음문법을사용하면튜플을두와이어, 이경우 equ 와 gt 로직접분해할수있습니다.

```
val (equ, gt) = compare(inA, inB)
```

함수는 Module 의일부로선언할수있습니다. 하지만여러모듈에서사용될함수는유틸리티함수를모아놓은 Scala 객체에두는것이더좋습니다.

10.3 조합논리생성하기

논리테이블 (진리표) 은조합논리입니다. 이는읽기전용메모리 (ROM) 라고도불리며, 알수있듯이테이블에대한입력은그러한 ROM 으로의주소입니다. VecInit 로논리테이블을생성합니다. 다음코드조각은숫자 n 의제곱을계산하는테이블을생성합니다.

```
val squareROM = VecInit(0.U, 1.U, 4.U, 9.U, 16.U, 25.U)
val square = squareROM(n)
```

Scala 의모든힘을활용하여논리 (테이블) 를생성할수있습니다. 예를들어, 삼각함수를나타내기위한고정소수점상수테이블을생성하거나, 디지털필터를위한상수를계산하거나, Chisel 로작성된마이크로프로세서를위한코드를생성하는어셈블러를 Scala 로작성할수있습니다. 이모든함수는동일한코드베이스 (동일한언어) 에있으며하드웨어생성중에실행될수있습니다.

테이블생성의고전적인예는이진수를 이진화십진(BCD) 표현으로변환하는것입니다. BCD 는각십진자릿수마다 4 비트를사용하여숫자를십진형식으로나타냅니다. 예를들어, 십진수 13 은이진수로 1101 이며, BCD 로는 1 과 3 을이진수로인코딩하여 00010011 이됩니다. BCD 는 16 진수보다사용자친화적인숫자

```
import chisel3._

class BcdTable extends Module {
  val io = IO(new Bundle {
    val address = Input(UInt(8.W))
    val data = Output(UInt(8.W))
  })

  val table = Wire(Vec(100, UInt(8.W)))

  // Convert binary to BCD
  for (i <- 0 until 100) {
    table(i) := (((i/10)<<4) + i%10).U
  }

  io.data := table(io.address)
}
```

Listing 10.1: 이진수를이진화십진수로변환하기.

표현인십진수로숫자를표시할수있게해줍니다.

Verilog 나 VHDL 같은고전적인하드웨어기술언어를사용할때는, 그러한테이블을생성하기위해다른스크립팅언어나프로그래밍언어를사용해야할것입니다. 이진수를 BCD 로변환하는테이블을계산하는 Java 프로그램을작성할수있습니다. 그 Java 프로그램은프로젝트에포함될수있는 VHDL 코드를출력합니다. 그 Java 프로그램은약 100 줄의코드이며, 그대부분이 VHDL 문자열을생성하는코드입니다. 하지만변환의핵심부분은단두줄의코드에불과합니다. Chisel 을사용하면, 하드웨어생성의일부로서이테이블을직접계산할수있습니다. 목록 10.1는이진수에서 BCD 로의변환을위한테이블생성을보여줍니다.

10.3.1 파일읽기

Scala Array 로부터논리테이블을생성할수도있습니다. 논리테이블을위해하드웨어생성시점에읽어들이고자하는데이터가파일에있을수있습니다. 목록 10.2는텍스트표현으로정수상수를담고있는 data.txt 파일을읽기위해 Scala

```
import chisel3._
import scala.io.Source

class FileReader extends Module {
  val io = IO(new Bundle {
    val address = Input(UInt(8.W))
    val data = Output(UInt(8.W))
  })

  val array = new Array[Int](256)
  var idx = 0

  // read the data into a Scala array
  val source = Source.fromFile("data.txt")
  for (line <- source.getLines()) {
    array(idx) = line.toInt
    idx += 1
  }

  // convert the Scala integer array to a Seq
  // and then into a vector of Chisel UInt
  val table = VecInit(array.toIndexedSeq.map(_.U(8.W)))

  // use the table
  io.data := table(io.address)
}
```

Listing 10.2: 텍스트파일을읽어논리테이블을생성하기.

표준라이브러리의 Scala Source 클래스를사용하는방법을보여줍니다.[3]
다소위압적으로보일수있는다음표현식에대해몇마디덧붙입니다:

```
val table = VecInit(array.toIndexedSeq.map(_.U(8.W)))
```

toIndexedSeq 메서드는 Scala Array 를매핑함수 map 을지원하는 Scala 시퀀스 (Seq) 로변환합니다. map 은시퀀스의각원소에대해함수를호출하고, 그함수의반환값들로이루어진시퀀스를반환합니다. 저희의함수 _.U(8.W) 는 Scala 배열의각 Int 값을 _ 로나타내며, Scala Int 값을 8 비트크기의 Chisel UInt 리터럴로변환합니다. Chisel 객체 VecInit 은 Chisel 타입의시퀀스 Seq 로부터 Chisel Vec 을생성합니다.

Chisel 의 Vec 을 Scala 시퀀스로부터초기화하는방법을사용하면직렬포트로내보낼수있는메시지를표현할수있습니다. Scala/Java String 은 Scala Seq 입니다. 따라서각 Scala Char 를 Chisel UInt 로매핑하는데 map 메서드를사용할수있습니다. 다음코드는 msg 문자열의표준인사말을 Chisel Vec 으로변환합니다:

```
val msg = "Hello World!"
val text = VecInit(msg.map(_.U))
val len = msg.length.U
```

이코드는직렬포트예제에서발췌한것이며, 이예제는이책의뒷부분에서환영메시지를보내는데사용됩니다.

10.3.2 타입변환

때로는한 Chisel 타입을다른타입으로변환하는것이편리할때가있습니다. 모든타입은비트의집합을표현합니다. 따라서이러한매핑을쉽게수행할수있습니다. 첫번째예로, 데이터를바이트단위로수신하여 4 바이트를 32 비트 UInt 로재포장하고자한다고가정해봅시다. 다음코드는바이트벡터를 UInt 로매핑합니다. 바이트는다음순서로 UInt 에매핑됩니다: 첫번째바이트는하위 8 비트에, 다음바이트는그다음 8 비트에, 이런식으로계속됩니다.

```
val vec = Wire(Vec(4, UInt(8.W)))
val word = vec.asUInt
```

[3] Scala 는 Java 를기반으로하므로, 예를들어이진파일을읽는등모든 Java 파일읽기및쓰기라이브러리를사용할수있습니다.

UInt word 가주어지면, 이를다시네개의바이트로이루어진벡터로변환할수있습니다.

```
val vec2 = word.asTypeOf(Vec(4, UInt(8.W)))
```

Bundle 을 UInt 로변환할수도있습니다. 번들필드는마지막필드 (이예제에서는필드 b) 가 UInt 의하위비트 (여기서는 15 부터 0 까지) 에오고, 그다음으로끝에서두번째필드가오는식으로순서가정해집니다. 이순서는 Vec 을매핑하는순서와반대라는점에유의하십시오.

```
class MyBundle extends Bundle {
  val a = UInt(8.W)
  val b = UInt(16.W)
}

val bundle = Wire(new MyBundle)
val word2 = bundle.asUInt
```

UInt 는다음과같이 (다시) 번들로변환할수있습니다:

```
val bundle2 = word2.asTypeOf(new MyBundle)
```

이변환은번들의모든필드를 0 으로초기화하는데에도사용할수있습니다.

```
val bundle3 = 0.U.asTypeOf(new MyBundle)
```

10.4 파라미터를이용한구성

Chisel 컴포넌트와함수는파라미터로구성할수있습니다. 파라미터는정수상수처럼단순할수도있지만, Chisel 하드웨어타입이될수도있습니다.

10.4.1 단순파라미터

회로를파라미터화하는가장간단한방법은비트폭을파라미터로정의하는것입니다. 파라미터는 Chisel 모듈의생성자에인자로전달할수있습니다. 다음은구성가능한비트폭을가진가산기를구현하는모듈의간단한예제입니다. 비트폭 n 은생성자에전달되는컴포넌트의파라미터 (Scala 타입 Int) 이며, IO 번들에서사용할수있습니다.

```
class ParamAdder(n: Int) extends Module {
  val io = IO(new Bundle{
    val a = Input(UInt(n.W))
    val b = Input(UInt(n.W))
    val c = Output(UInt(n.W))
  })

  io.c := io.a + io.b
}
```

가산기의파라미터화된버전은다음과같이생성할수있습니다:

```
  val add8 = Module(new ParamAdder(8))
  val add16 = Module(new ParamAdder(16))
```

10.4.2 케이스클래스

더많은파라미터가필요한경우, Chisel 모듈의생성자에파라미터를추가로넣을수있습니다. 하지만이러한파라미터를여러생성자를거쳐전달하다보면사용하기가번거로워질수있습니다. 게다가파라미터의개수나타입을변경할때여러곳을수정해야합니다.

Scala 에는여러필드를클래스로묶는매우가벼운구성요소가있는데, 바로 케이스클래스입니다. 케이스클래스는일반 Scala 클래스와비슷하지만정의가매우가볍습니다. 다음코드는세개의파라미터를표현하는케이스클래스를정의합니다. 이는특정폭의송신 (tx) 버퍼와수신 (rx) 버퍼를가진장치에사용될수있습니다.

```
case class Config(txDepth: Int, rxDepth: Int, width: Int)
```

이케이스클래스의객체는단순히생성자를호출하기만하면생성됩니다. 필드는불변이며접근하여읽을수있습니다:

```
  val param = Config(4, 2, 16)

  println("The width is " + param.width)
```

매개변수가유효한지확인하는코드를케이스클래스에추가할수도있습니다.

```
case class SaveConf(txDepth: Int, rxDepth: Int, width: Int) {
```

```
  assert(txDepth > 0 && rxDepth > 0 && width > 0,
      "parameters must be larger than 0")
}
```

10.4.3 타입매개변수를가진함수

비트폭을설정매개변수로갖는것은하드웨어생성기의출발점에불과합니다. Chisel 의타입매개변수는매우유연한설정을가능하게합니다. 이기능덕분에 Chisel 은어떤종류의멀티플렉싱이든받아들일수있는멀티플렉서 (Mux) 를제공할수있습니다. 설정을위해 Chisel 타입을사용하는방법을보여주기위해임의의타입을받아들이는멀티플렉서를만들어보겠습니다. 다음함수는이멀티플렉서를정의합니다:

```
def myMux[T <: Data](sel: Bool, tPath: T, fPath: T): T = {

  val ret = WireDefault(fPath)
  when (sel) {
    ret := tPath
  }
  ret
}
```

Chisel 에서는함수를타입으로, 우리의경우에는 Chisel 타입으로매개변수화할수있습니다. 대괄호안의표현식 [T <: Data] 는 Data 이거나 Data 의하위클래스인타입매개변수 T 를정의합니다. Data 는 Chisel 타입시스템의루트입니다.

이멀티플렉서함수는세개의매개변수를가지고있습니다: 불리언조건, 참경로를위한매개변수하나, 그리고거짓경로를위한매개변수하나입니다. 두경로매개변수모두타입 T 를가지며, 함수호출시에제공됩니다. 함수자체는간단합니다: fPath 의기본값을가진와이어를정의하고, 조건이참이면값을 tPath 로변경합니다. 이조건은전형적인멀티플렉서함수입니다. 함수의마지막에서멀티플렉서하드웨어 (출력) 를반환합니다. 이멀티플렉서함수를 UInt 와같은단순한타입과함께사용할수있습니다:

```
val resA = myMux(selA, 5.U, 10.U)
```

두멀티플렉서경로의타입은동일해야합니다. 다음의잘못된멀티플렉서사용은런타임오류를발생시킵니다:

```
  val resErr = myMux(selA, 5.U, 10.S)
```

더복잡한멀티플렉서를보여주기위해, 두개의필드를가진 Bundle 로새로운타입을정의합니다:

```
class ComplexIO extends Bundle {
  val d = UInt(10.W)
  val b = Bool()
}
```

먼저 Bundle 의 Wire 를생성한다음하위필드를설정하여 Bundle 상수를정의할수있습니다. 그러면이복잡한타입으로매개변수화된멀티플렉서를사용할수있습니다.

```
  val tVal = Wire(new ComplexIO)
  tVal.b := true.B
  tVal.d := 42.U
  val fVal = Wire(new ComplexIO)
  fVal.b := false.B
  fVal.d := 13.U

  // The multiplexer with a complex type
  val resB = myMux(selB, tVal, fVal)
```

함수의초기설계에서는 WireDefault 를사용하여기본값을가진타입 T 의와이어를생성했습니다. 기본값을사용하지않고 Chisel 타입만을가진와이어를생성해야한다면, fPath.cloneType 을사용하여 Chisel 타입을얻을수있습니다. 다음함수는멀티플렉서를코딩하는대안적인방법을보여줍니다.

```
  def myMuxAlt[T <: Data](sel: Bool, tPath: T, fPath: T): T
      = {

    val ret = Wire(fPath.cloneType)
    ret := fPath
    when (sel) {
      ret := tPath
    }
    ret
  }
```

10.4.4 타입매개변수를가진모듈

모듈또한 Chisel 타입으로매개변수화할수있습니다. 서로다른처리코어간에데이터를이동시키기위한네트워크온칩을설계하고자한다고가정해보겠습니다. 다만라우터인터페이스에데이터형식을하드코딩하고싶지않고, 이를 매개변수화하고자합니다. 함수의타입매개변수와유사하게, Module 생성자에타입매개변수 T 를추가합니다. 또한그타입의생성자매개변수를하나가져야합니다. 이예제에서는라우터포트의수도설정가능하게만듭니다.

```
class NocRouter[T <: Data](dt: T, n: Int) extends Module {
  val io =IO(new Bundle {
    val inPort = Input(Vec(n, dt))
    val address = Input(Vec(n, UInt(8.W)))
    val outPort = Output(Vec(n, dt))
  })

  // Route the payload according to the address
  // ...
```

라우터를사용하려면, 먼저라우팅하려는데이터타입을예를들어 Chisel Bundle 로정의해야합니다:

```
class Payload extends Bundle {
  val data = UInt(16.W)
  val flag = Bool()
}
```

사용자정의번들의인스턴스와포트수를라우터의생성자에전달하여라우터를생성합니다:

```
  val router = Module(new NocRouter(new Payload, 2))
```

10.4.5 매개변수화된 Bundle

라우터예제에서는라우터의입력에대해두개의서로다른필드벡터를사용했습니다. 하나는주소용이고다른하나는매개변수화된데이터용이었습니다. 더우아한해결책은그자체가매개변수화된 Bundle 을갖는것입니다. 다음과같은형태입니다:

```
class Port[T <: Data](dt: T) extends Bundle {
```

```
  val address = UInt(8.W)
  val data = dt.cloneType
}
```

이 Bundle 은 T 타입의매개변수를가지며, 이는 Chisel 의 Data 타입의서브타입입니다. 번들내부에서는이매개변수에대해 cloneType 을호출하여 data 필드를정의합니다. 그러나생성자매개변수를사용하면이매개변수는클래스의공개 (public) 필드가됩니다. Chisel 이 Bundle 의타입을복제해야할때, 예를들어 Vec 에서사용될때, 이공개필드가방해가됩니다. 이문제에대한해법 (우회책) 은매개변수필드를비공개 (private) 로만드는것입니다.

```
class Port[T <: Data](private val dt: T) extends Bundle {
  val address = UInt(8.W)
  val data = dt.cloneType
}
```

이새로운 Bundle 로, 우리는라우터포트를정의할수있습니다

```
class NocRouter2[T <: Data](dt: T, n: Int) extends Module {
  val io =IO(new Bundle {
    val inPort = Input(Vec(n, dt))
    val outPort = Output(Vec(n, dt))
  })

  // Route the payload according to the address
  // ...
```

그리고 Payload 를매개변수로받는 Port 로그라우터를인스턴스화합니다:

```
  val router = Module(new NocRouter2(new Port(new Payload),
      2))
```

10.4.6 선택적포트

일부하드웨어생성기는구성에따라달라지는 IO 포트를가질수있습니다. 예를들어, 전형적인 32 비트 RISC 프로세서를위한레지스터파일을구현한다고합시다. 디버깅을위해서는모든레지스터에접근할수있기를원합니다. 따라서테스터에서모든레지스터를읽을수있는추가포트를두고자합니다. 그러나 Verilog 생성시에는이비용이큰추가포트를원하지않습니다.

목록 10.3은이러한레지스터파일을보여줍니다. 이레지스터파일은 Scala

```
class RegisterFile(debug: Boolean) extends Module {
  val io = IO(new Bundle {
    val rs1 = Input(UInt(5.W))
    val rs2 = Input(UInt(5.W))
    val rd = Input(UInt(5.W))
    val wrData = Input(UInt(32.W))
    val wrEna = Input(Bool())
    val rs1Val = Output(UInt(32.W))
    val rs2Val = Output(UInt(32.W))
    val dbgPort = if (debug)
      Some(Output(Vec(32, UInt(32.W)))) else None
  })
  val regfile = RegInit(VecInit(Seq.fill(32)(0.U(32.W))))
  io.rs1Val := regfile(io.rs1)
  io.rs2Val := regfile(io.rs2)
  when(io.wrEna) {
    regfile(io.rd) := io.wrData
  }
  if (debug) {
    io.dbgPort.get := regfile
  }
}
```

Listing 10.3: 선택적디버그포트를가진레지스터파일입니다.

Boolean 인 debug 를매개변수로가집니다. IO 번들의정의에서는이매개변수를사용하여해당포트를생성할지여부를결정합니다. Scala 에서는이것이 Option 으로표현됩니다. 옵션은 Some 으로감싼값을반환하거나, 값이없음을 None 으로나타낼수있습니다. 모듈의본문에서는 get 으로옵션의값을추출합니다.

테스터에서도마찬가지로 get 메서드를사용하여해당 IO 포트에대한참조를얻어야합니다.

```
dut.io.dbgPort.get(4).expect(123.U)
```

```
abstract class Ticker(n: Int) extends Module {
  val io = IO(new Bundle{
    val tick = Output(Bool())
  })
}
```

Listing 10.4: 저희의 ticker 구현을위한기본클래스입니다.

```
class UpTicker(n: Int) extends Ticker(n) {

  val N = (n-1).U

  val cntReg = RegInit(0.U(8.W))

  cntReg := cntReg + 1.U
  val tick = cntReg === N
  when(tick) {
    cntReg := 0.U
  }

  io.tick := tick
}
```

Listing 10.5: 카운터를이용한틱생성입니다.

10.5 상속의사용

Chisel 은객체지향언어입니다. 하드웨어컴포넌트인 Chisel Module 은 Scala 클래스입니다. 따라서공통동작을부모클래스로추출하기위해상속을사용할수있습니다. 예제를통해상속을어떻게사용하는지살펴보겠습니다.

Section 6.2에서저주파틱생성에사용할수있는다양한형태의카운터를살펴보았습니다. 예를들어자원요구량을비교하기위해이러한여러버전을살펴보고자한다고가정해봅시다. 틱킹인터페이스를정의하는추상클래스로시작하며, 이는 Listing 10.4에나타나있습니다.
목록 10.5은틱생성을위해카운터로위쪽으로카운트하는방식으로이추상클래스를구현한첫번째예를보여줍니다.

```
import chisel3._
import chiseltest._
import org.scalatest.flatspec.AnyFlatSpec

trait TickerTestFunc {
  def testFn[T <: Ticker](dut: T, n: Int) = {
    // -1 means that no ticks have been seen yet
    var count = -1
    for (_ <- 0 to n * 3) {
      // Check for correct output
      if (count > 0)
        dut.io.tick.expect(false.B)
      else if (count == 0)
        dut.io.tick.expect(true.B)

      // Reset the counter on a tick
      if (dut.io.tick.peekBoolean())
        count = n-1
      else
        count -= 1
      dut.clock.step()
    }
  }
}
```

Listing 10.6: 티커의여러버전을위한테스터.

저희는단일테스트벤치로 티커로직의모든서로다른버전을테스트할수있습니다. 단지테스트벤치가 Ticker 의서브타입을받아들이도록정의하기만하면됩니다. 목록 10.6는테스터를위한 Chisel 코드를보여줍니다. TickerTester 는여러매개변수를가집니다: (1) Ticker 또는 Ticker 를상속하는모든클래스를받아들이기위한타입매개변수 [T <: Ticker], (2) T 타입또는그서브타입인, 테스트대상설계, 그리고 (3) 각틱마다기대되는클록사이클수입니다. 테스터는틱이처음발생하는시점을기다린다음 (구현에따라시작시점이다를수있습니다) tick 이매 n 클록사이클마다반복되는지확인합니다.

티커의첫번째간단한구현으로, 아마도 println 디버깅을사용하여테스터자체를테스트할수있습니다. 단순한티커와테스터가올바르다고확신할때, 티커의두가지버전을더탐구해볼수있습니다. 목록 10.7은 0 까지카운트다운하는카운터

```
class DownTicker(n: Int) extends Ticker(n) {

  val N = (n-1).U

  val cntReg = RegInit(N)

  cntReg := cntReg - 1.U
  when(cntReg === 0.U) {
    cntReg := N
  }

  io.tick := cntReg === N
}
```

Listing 10.7: 다운카운터를사용한틱생성.

를사용한틱생성을보여줍니다. 목록 10.8는비교기를피함으로써더적은하드웨어를사용하기위해 -1 까지카운트다운하는너드버전을보여줍니다.

ScalaTest 명세를사용하여티커의여러버전에대한인스턴스를생성하고이를범용테스트벤치에전달함으로써티커의세가지버전을모두테스트할수있습니다. 목록 10.9은테스트명세를보여줍니다. 다음명령으로티커테스트만실행합니다:

```
sbt "testOnly TickerTest"
```

10.6 함수형프로그래밍을이용한하드웨어생성

Scala 는함수형프로그래밍을지원하므로, Chisel 도마찬가지입니다. 함수를사용하여하드웨어를표현할수있으며, " 고차함수" 를사용하여함수형프로그래밍으로이러한하드웨어컴포넌트들을결합할수있습니다. 간단한예시인벡터의합부터시작해봅시다:

```
  def add(a: UInt, b:UInt) = a + b

  val sum = vec.reduce(add)
```

먼저, 함수 add 안에가산기용하드웨어를정의합니다. 벡터 (Chisel 타입 Vec) 는 vec 에위치합니다. Scala 메소드 reduce() 는이항연산으로모든컬렉션요소

```
class NerdTicker(n: Int) extends Ticker(n) {

  val N = n

  val MAX = (N - 2).S(8.W)
  val cntReg = RegInit(MAX)
  io.tick := false.B

  cntReg := cntReg - 1.S
  when(cntReg(7)) {
    cntReg := MAX
    io.tick := true.B
  }
}
```

Listing 10.8: -1 까지카운트다운하여생성한틱.

```
class TickerTest extends AnyFlatSpec with
    ChiselScalatestTester with TickerTestFunc {
  "UpTicker 5" should "pass" in {
    test(new UpTicker(5)) { dut => testFn(dut, 5) }
  }

  "DownTicker 7" should "pass" in {
    test(new DownTicker(7)) { dut => testFn(dut, 7) }
  }

  "NerdTicker 11" should "pass" in {
    test(new NerdTicker(11)) { dut => testFn(dut, 11) }
  }
}
```

Listing 10.9: 티커테스트를위한 ChiselTest.

를결합하여하나의값을산출합니다. reduce() 메소드는첫번째요소부터시작하여 시퀀스를축소합니다. 처음두요소를취하여연산을수행합니다. 그결과는단하나 의결과만남을때까지다음요소와결합됩니다.

두요소를결합하는함수는 reduce 에매개변수로제공되며, 우리의경우에는가 산기를반환하는 add 입니다. 결과로생성되는하드웨어는벡터 vec 요소들의합을 계산하는가산기의체인입니다. (단순한) add 함수를정의하는대신, 덧셈을익명 함수로제공하고두피연산자를나타내기위해 Scala 와일드카드"_" 를사용할수 있습니다.

```
val sum = vec.reduce(_ + _)
```

이한줄의코드로가산기체인을생성했습니다. 합함수의경우체인이이상적이지는 않습니다. 트리를사용하면조합지연이더짧아집니다. 합성도구가우리의가산기 체인을재배열할것이라고신뢰하지못하는경우, Chisel 의 reduceTree 메소드를 사용하여가산기트리를생성할수있습니다:

```
val sum = vec.reduceTree(_ + _)
```

10.6.1 최솟값탐색예제

더정교한예제로서, Vec 에서최솟값을찾는회로를만들어보겠습니다. 이회로를 표현하기위해 Scala 에서 함수리터럴이라불리는익명함수를사용합니다. 함수 리터럴의구문은괄호안의매개변수다음에 => 가오고, 그다음함수본문이오는형 태입니다:

```
(param) => function body
```

최솟값함수를위한함수리터럴은두매개변수 x 와 y 를사용하며, 두매개변수를 비교하여더작은값을반환하는멀티플렉서 (Mux) 를반환합니다.

```
val min = vec.reduceTree((x, y) => Mux(x < y, x, y))
```

이회로를확장하여 vec 에서최솟값뿐만아니라그최솟값의 vec 내위치 (인덱 스) 도반환하도록해봅시다. 두값을반환하기위해값과인덱스를담을번들 Two 를 정의합니다. 이번들들을담을수있는 vecTwo Vec 을선언하고, 목록 10.10에서 보듯이이를루프내에서원래의입력및 Vec 내인덱스와연결합니다.

이전과마찬가지로, vecTwo 의 reduceTree 메소드에서함수리터럴을사용하여 번들내의값필드를비교하고전체번들에대한멀티플렉서를반환합니다. 값 res 는 최솟값과위치를담고있는번들을가리킵니다.

```
class Two extends Bundle {
  val v = UInt(w.W)
  val idx = UInt(8.W)
}

val vecTwo = Wire(Vec(n, new Two()))
for (i <- 0 until n) {
  vecTwo(i).v := vec(i)
  vecTwo(i).idx := i.U
}

val res = vecTwo.reduceTree((x, y) => Mux(x.v < y.v, x, y))
```

Listing 10.10: 인덱스를포함한최솟값탐색.

최솟값탐색회로의더발전된변형으로서, 값과인덱스를반환하기위한번들을생성하지않도록더많은 Scala 기능을사용해보겠습니다. 두값을나타내기위해튜플을사용할것입니다. 다음코드는원래시퀀스에함수체인을적용하는것을보여줍니다. 함수를체이닝하는것은함수형프로그래밍에서전형적인패턴입니다. 이패턴은연산들의파이프라인으로도볼수있습니다.

TODO: 예시들사이의순서를올바르게맞추려면이는리스팅에있어야합니다.

```
val resFun = vec.zipWithIndex
  .map((x) => (x._1, x._2.U))
  .reduce((x, y) => (Mux(x._1 < y._1, x._1, y._1),
      Mux(x._1 < y._1, x._2, y._2)))
```

첫번째함수 (zipWithIndex) 는원래의 UInt 시퀀스를튜플의시퀀스로변환하는데, 첫번째요소는변경되지않은 UInt 이고두번째요소는 Scala Int 로서 vec 내에서의인덱스값입니다. 일반적으로 zip 함수는두시퀀스를병합 (zip) 하여두요소를튜플로담은하나의시퀀스로만듭니다.

다음함수는 Chisel UInt 와 Scala Int 로이루어진튜플을두개의 Chisel UInt 로매핑합니다. reduce 함수는최솟값찾기의생성을제공합니다. 우리는두개의멀티플렉서에서튜플의첫번째요소를비교하고, 최솟값과위치를 Chisel UInt 타입으로담은튜플을반환합니다.

전체함수형표현식은중간결과를담기위해 Scala Vector 를사용하지만, Chisel 타입만으로구성된하드웨어 (연결된멀티플렉서들) 를반환한다는점에유의하십시오. 여기서는 Scala Vector 를사용하므로, Chisel 의 Vec 에서만

사용가능한 reduceTree 를사용할수없습니다.

reduceTree 를계속사용하기위해, 다음해법은 Scala 튜플대신 Chisel MixedVec 을사용합니다. Chisel MixedVec 은서로다른위치에서로다른타입을가질수있다는점에서 Scala 튜플과유사합니다. 따라서이는예를들어멀티플렉서로서기능할수는없습니다. 하지만하드웨어생성도중인덱싱가능한컬렉션으로는사용할수있습니다.

```
val scalaVector = vec.zipWithIndex
  .map((x) => MixedVecInit(x._1, x._2.U(8.W)))
val resFun2 = VecInit(scalaVector)
  .reduceTree((x, y) => Mux(x(0) < y(0), x, y))

val minVal = resFun2(0)
val minIdx = resFun2(1)
```

위예시에서는값과그인덱스로이루어진 Scala Vector 를생성하되, 이번에는 Chisel 의" 튜플" 을사용합니다. 그런다음이 Scala Vector 를 Chisel Vec 으로변환합니다. 그러면다시트리기반축소 (reduction) 를수행할수있습니다. 이버전의또다른이점은멀티플렉서가단하나뿐이라는점인데, 이멀티플렉서는실제로는 MixedVec 인두개의 Chisel " 튜플" 사이에서선택합니다. resFun2 의결과는두요소를가진 MixedVec 이며, " 일반" Vec 과마찬가지로인덱스로접근합니다.

10.6.2 아비트레이션트리

트리축소함수를이용하면오직 2:1 아비터만으로아비트레이션트리를구축할수있습니다. 아비트레이션회로는다음과같이생성할수있습니다:

```
class Arbiter[T <: Data: Manifest](n: Int, private val gen:
    T) extends Module {
  val io = IO(new Bundle {
    val in = Flipped(Vec(n, new DecoupledIO(gen)))
    val out = new DecoupledIO(gen)
  })

  io.out <> io.in.reduceTree((a, b) => arbitrateSimp(a, b))
}
```

입력은 ready/valid 인터페이스의 Vec 이고출력은단일 ready/valid 인터페

이스입니다. 우리에게 필요한것은단지두요청사이의아비트레이션을제공하는 함수뿐입니다.

단순아비트레이션

첫번째해법으로, 절 10.6.2에서보인아비터와유사한우선순위기반아비트레이션을구축하겠습니다. 절 10.6.2의그아비터는순수하게조합회로였습니다. 하지만 ready/valid 인터페이스에서는 ready 신호와 valid 신호사이에조합적인흐름이있어서는안됩니다. 따라서이신호들에대한상태를도입하고들어오는데이터를레지스터에저장해야합니다.

Listing 10.11는 2 대 1 아비트레이션함수를보여줍니다. 이함수는 valid 를어서트한요청자가수신자에의해읽힐때 (ready 로신호됨) 에만이를디어서트한다고가정합니다. 또한, 현재클록사이클의 valid 에따라다음클록사이클에 ready 를설정할수있습니다. 이는 ready/valid 프로토콜의한가지특정해석으로, AXI 에서도사용됩니다.

다음과같은레지스터가필요합니다: 출력을위한데이터를담을 regData, 데이터레지스터가비어있음을신호하는플래그인 regEmpty, 그리고두입력의 ready 신호를위한두개의플래그 (regReadyA 와 regReadyB) 입니다. 함수의반환값 (out) 은 DecoupledIO 타입의와이어입니다.

데이터레지스터가비어있고두입력중하나가 valid 입력을신호할때, 우리는다음클록사이클에 (ready 레지스터를통해) ready 를신호합니다. 아비터는데이터레지스터를하나만가지고있으므로, 두입력중하나에만아비터가 ready 임을신호할수있다는점에유의하십시오. 등록된 ready 를갖게되면, 입력이여전히 valid 하다고가정하고데이터를등록하며, regEmpty 를해제하고 ready 플래그를리셋합니다.

데이터레지스터가비어있지않을때출력은 valid 입니다. 수신자가 ready 일때데이터가전송되고데이터레지스터는다시비게됩니다. 마지막문장으로서, 함수는 DecoupledIO 와이어에대한참조인 out 을반환합니다.

공정한아비트레이션

절 10.6.2에서우리는아비트레이션회로의조합적버전을제시했습니다. 이조합적버전은우선순위아비터이므로, 우선순위가높은하나의요청자가아비트레이션을독점할수있습니다. 이러한독점을피하려면, 지난번에누가아비트레이션에서이겼는지기억할상태를도입해야합니다. 우리의가정은 2:1 아비터가공정하다면, 균형잡힌아비트레이션트리에서도공정한아비트레이션이이루어진다는것입니다.

```
  def arbitrateSimp(a: DecoupledIO[T], b: DecoupledIO[T]) = {

    val regData = Reg(gen)
    val regEmpty = RegInit(true.B)
    val regReadyA = RegInit(false.B)
    val regReadyB = RegInit(false.B)

    val out = Wire(new DecoupledIO(gen))

    when (a.valid & regEmpty & !regReadyB) {
      regReadyA := true.B
    } .elsewhen (b.valid & regEmpty & !regReadyA) {
      regReadyB := true.B
    }
    a.ready := regReadyA
    b.ready := regReadyB

    when (regReadyA) {
      regData := a.bits
      regEmpty := false.B
      regReadyA := false.B
    }
    when (regReadyB) {
      regData := b.bits
      regEmpty := false.B
      regReadyB := false.B
    }

    out.valid := !regEmpty
    when (out.ready) {
      regEmpty := true.B
    }

    out.bits := regData
    out
  }
```

Listing 10.11: 2 대 1 단순아비터.

```
  def arbitrateFair(a: DecoupledIO[T], b: DecoupledIO[T]) = {
    object State extends ChiselEnum {
      val idleA, idleB, hasA, hasB = Value
    }
    import State._
    val regData = Reg(gen)
    val regState = RegInit(idleA)
    val out = Wire(new DecoupledIO(gen))
    a.ready := regState === idleA
    b.ready := regState === idleB
    out.valid := (regState === hasA || regState === hasB)
    switch(regState) {
      is (idleA) {
        when (a.valid) {
          regData := a.bits
          regState := hasA
        } otherwise {
          regState := idleB
        }
      }
      is (idleB) {
        when (b.valid) {
          regData := b.bits
          regState := hasB
        } otherwise {
          regState := idleA
        }
      }
      is (hasA) {
        when (out.ready) {
          regState := idleB
        }
      }
      is (hasB) {
        when (out.ready) {
          regState := idleA
        }
      }
    }
    out.bits := regData
    out
  }
```

Listing 10.12: 공정한 2 대 1 아비터.

리스팅 10.12은그공정한 2:1 아비트레이션회로를보여줍니다. 이아비터는저장할데이터를위한레지스터하나와상태레지스터하나를포함합니다. 공정성을위해, 요청이없을때아비터는두개의유휴상태 (idleA 와 idleB) 사이를전환합니다. 두유휴상태각각은두입력중하나만을받아들입니다. 저장을위한레지스터가단하나뿐이므로, 아비터는두입력중하나에대해서만 ready 일수있다는점에유의하십시오. 두입력모두에대해 ready 이면서우선순위를전환할수있게하려면, 같은클록사이클에두입력이모두 valid 인경우를처리할두번째데이터레지스터가필요할것입니다.

요청이받아들여지면, 그데이터를저장하고완료상태중하나 (hasA 또는 hasB) 로전환합니다. 출력의소비자가데이터를받아들이면, 아비터는유휴상태로다시전환됩니다. 이때아비터는다음클록사이클에다른입력의대기중인요청을받아들일유휴상태로전환합니다.

11 예제설계

이절에서는더큰설계의구성요소로사용되는 FIFO 버퍼와같은몇가지소규모디지털설계를살펴봅니다. 또다른예로, FIFO 버퍼를사용할수있는직렬인터페이스 (UART 라고도함) 를설계합니다. 나아가 FIFO 인터페이스를일반화하여다양한구현방법을보여줄것입니다.

11.1 FIFO 버퍼

라이터 (송신자) 와리더 (수신자) 는라이터와리더사이에위치한버퍼로분리할수있습니다. 일반적인버퍼는선입선출 (FIFO) buffer) 입니다. 그림 11.1는라이터, FIFO, 리더를보여줍니다. 라이터는활성화된 write 신호와함께 din 을통해데이터를 FIFO 에넣습니다. 데이터는활성화된 read 신호와함께리더에의해 dout 을통해 FIFO 로부터읽힙니다.

FIFO 는초기에비어있으며, 이는 empty 신호로표시됩니다. 빈 FIFO 에서읽는것은일반적으로정의되어있지않습니다. 데이터가기록되고전혀읽히지않으면 FIFO 는 full 상태가됩니다. 가득찬 FIFO 에쓰기를하면일반적으로무시되며데이터는손실됩니다. 다시말해, empty 와 full 신호는핸드셰이크신호역할을합니다.

FIFO 는여러방식으로구현할수있습니다. 예를들어온칩메모리와읽기·쓰기포인터를사용하거나, 단순히작은상태기계를갖춘레지스터의체인을사용할수있습니다. 소규모버퍼 (수십개원소이하) 의경우, 개별레지스터를체인형태로연결

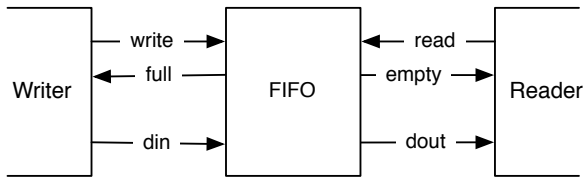


Figure 11.1: 작성자, FIFO 버퍼, 그리고판독자.

하여구성한 FIFO 는자원요구량이낮은단순한구현방식입니다. 버블 FIFO 의 코드는 chisel-examples 저장소에서확인할수있습니다.[1]

먼저작성자측과판독자측의 IO 신호를정의하는것으로시작합니다. 데이터크기는 size 로설정할수있습니다. 쓰기데이터는 din 이며, 쓰기는 write 로신호를보냅니다. full 신호는작성자측에서 흐름제어를수행합니다.

```
class WriterIO(size: Int) extends Bundle {
  val write = Input(Bool())
  val full = Output(Bool())
  val din = Input(UInt(size.W))
}
```

판독자측은 dout 으로데이터를제공하며, 읽기는 read 로시작됩니다. empty 신호는판독자측의흐름제어를담당합니다.

```
class ReaderIO(size: Int) extends Bundle {
  val read = Input(Bool())
  val empty = Output(Bool())
  val dout = Output(UInt(size.W))
}
```

목록 11.1는단일버퍼를보여줍니다. 이버퍼는 WriterIO 타입의인큐잉포트 enq 와 ReaderIO 타입의디큐잉포트 deq 를가지고있습니다. 버퍼의상태요소는 데이터를보관하는레지스터하나 (dataReg) 와단순한 FSM 을위한상태레지스터하나 (stateReg) 입니다. 이 FSM 은두가지상태만을가집니다. 즉버퍼가 empty 이거나 full 인상태입니다. 버퍼가 empty 상태일때쓰기를하면입력데이터를레지스터에저장하고 full 상태로전이합니다. 버퍼가 full 상태일때읽기를하면데이터를소비하고 empty 상태로전이합니다. IO 포트 full 과 empty 는작성자와판독자에게버퍼상태를나타냅니다.

목록 11.2는완전한 FIFO 를보여줍니다. 완전한 FIFO 는개별 FIFO 버퍼와동일한 IO 인터페이스를가집니다. BubbleFifo 는매개변수로데이터워드의 size 와버퍼단계수를나타내는 depth 를가집니다. depth 개의 FifoRegister 로 depth 단계의버블 FIFO 를구성할수있습니다. 각단계는 Scala Array 에채워넣어생성합니다. Scala 배열자체는하드웨어적인의미를가지지않으며, 단지생성된버퍼들에대한참조를담는컨테이너역할을할뿐입니다. Scala for 루프안에서개별버퍼들을연결합니다. 첫번째버퍼의인큐잉측은완전한 FIFO 의인큐잉 IO 에연결되고, 마지막버퍼의디큐잉측은완전한 FIFO 의디큐잉측에연결됩니다.

[1] 완전성을위해, Chisel 책저장소에도 FIFO 코드사본이포함되어있습니다.

```
class FifoRegister(size: Int) extends Module {
  val io = IO(new Bundle {
    val enq = new WriterIO(size)
    val deq = new ReaderIO(size)
  })

  object State extends ChiselEnum {
    val empty, full = Value
  }
  import State._

  val stateReg = RegInit(empty)
  val dataReg = RegInit(0.U(size.W))

  when(stateReg === empty) {
    when(io.enq.write) {
      stateReg := full
      dataReg := io.enq.din
    }
  }.elsewhen(stateReg === full) {
    when(io.deq.read) {
      stateReg := empty
      dataReg := 0.U // just to better see empty slots in
          the waveform
    }
  }.otherwise {
    // There should not be an otherwise state
  }

  io.enq.full := (stateReg === full)
  io.deq.empty := (stateReg === empty)
  io.deq.dout := dataReg
}
```

Listing 11.1: 버블 FIFO 의단일단계.

```
class BubbleFifo(size: Int, depth: Int) extends Module {
  val io = IO(new Bundle {
    val enq = new WriterIO(size)
    val deq = new ReaderIO(size)
  })

  val buffers = Array.fill(depth) { Module(new
      FifoRegister(size)) }
  for (i <- 0 until depth - 1) {
    buffers(i + 1).io.enq.din := buffers(i).io.deq.dout
    buffers(i + 1).io.enq.write := ~buffers(i).io.deq.empty
    buffers(i).io.deq.read := ~buffers(i + 1).io.enq.full
  }
  io.enq <> buffers(0).io.enq
  io.deq <> buffers(depth - 1).io.deq
}
```

Listing 11.2: FIFO 는 FIFO 버블단계의배열로구성됩니다.

개별버퍼를연결하여 FIFO 큐를구현하는이방식은, 데이터가큐를거쳐거품처럼흘러간다는의미에서버블 FIFO 라고불립니다. 이는데이터전송률이클록속도보다상당히느릴때사용하기에단순하고좋은해결책입니다. 예를들어다음절에서소개할직렬포트의분리버퍼로사용할수있습니다.

그러나데이터전송률이클록주파수에근접하면버블 FIFO 는두가지한계를가집니다. (1) 각버퍼의상태가 비어있음와 가득참사이를전환해야하므로, FIFO 의최대처리량은워드당클록사이클 2 회가됩니다. (2) 데이터는완전한 FIFO 전체를거쳐거품처럼흘러가야하므로, 입력에서출력까지의지연시간은최소한버퍼의개수만큼걸립니다. FIFO 의다른가능한구현방식은절 11.3에서소개하겠습니다.

11.2 직렬포트

직렬포트 (UART 또는 RS-232라고도함) 는노트북과 FPGA 보드사이에서통신하는가장쉬운방법중하나입니다. 이름이나타내듯이데이터는직렬로전송됩니다. 8 비트바이트는다음과같은순서로전송됩니다. 즉시작비트 (0) 하나, 최하위비트부터시작되는 8 비트데이터, 그리고정지비트 (1) 하나또는두개입니다.

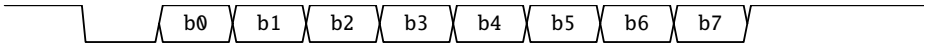


Figure 11.2: UART 가전송하는한바이트.

데이터가전송되지않을때출력은 1 입니다. 그림 11.2는한바이트가전송되는타이밍다이어그램을보여줍니다.

우리의 UART 는모듈마다최소한의기능만가지도록모듈식으로설계합니다. 송신기 (TX), 수신기 (RX), 버퍼를소개한다음, 이기본컴포넌트들의사용법을다룹니다.

먼저인터페이스와포트정의가필요합니다. UART 설계를위해서는데이터크기가 8 비트인 ready/valid 핸드셰이크인터페이스 (DecoupledIO 를확장) 를사용합니다.

```
class UartIO extends DecoupledIO(UInt(8.W)) {
}
```

ready/valid 인터페이스의관례는 ready 와 valid 가모두참일때데이터가전송된다는것입니다.

목록 11.3은최소한의구성요소만갖춘직렬송신기 (Tx) 를보여줍니다. IO 포트는직렬데이터가전송되는 txd 포트와, 송신기가직렬화하여전송할문자를수신할수있는 channel 입니다. 타이밍을생성하기위해, 직렬비트하나에해당하는시간을클록사이클단위의상수로계산합니다.

우리는세개의레지스터를사용합니다: (1) 데이터를시프트 (직렬화) 하기위한레지스터 (shiftReg), (2) 올바른보율 (baud rate) 을생성하기위한카운터 (cntReg), (3) 아직시프트아웃해야할비트수를위한카운터 (bitsReg) 입니다. 추가적인상태레지스터나 FSM 은필요하지않으며, 모든상태는이세개의레지스터에인코딩되어있습니다.

카운터 cntReg 는계속해서동작합니다 (0 까지카운트다운한후 0 이되면시작값으로다시로드됩니다). 모든동작은 cntReg 가 0 일때만수행됩니다. 최소한의송신기를구성하고있으므로, 데이터를저장하기위한시프트레지스터만존재합니다. 따라서채널은 cntReg 가 0 이고시프트아웃할비트가남아있지않을때만준비된 (ready) 상태가됩니다. IO 포트 txd 는시프트레지스터의최하위비트에직접연결됩니다.

시프트아웃할비트가더남아있을때 (bitsReg =/= 0.U), 비트를오른쪽으로시프트하고 1(송신기의유휴레벨) 로채웁니다. 더이상시프트아웃할비트가없다면, 채널에데이터가있는지확인합니다 (io.channel.valid 입력으로신호됩니다). 만약데이터가있다면, 시프트아웃할비트문자열은시작비트하나 (0), 8 비트데

```
class Tx(frequency: Int, baudRate: Int) extends Module {
  val io = IO(new Bundle {
    val txd = Output(UInt(1.W))
    val channel = Flipped(new UartIO())
  })

  val BIT_CNT = ((frequency + baudRate / 2) / baudRate -
    1).asUInt

  val shiftReg = RegInit(0x7ff.U)
  val cntReg = RegInit(0.U(20.W))
  val bitsReg = RegInit(0.U(4.W))

  io.channel.ready := (cntReg === 0.U) && (bitsReg === 0.U)
  io.txd := shiftReg(0)

  when(cntReg === 0.U) {

    cntReg := BIT_CNT
    when(bitsReg =/= 0.U) {
      val shift = shiftReg >> 1
      shiftReg := 1.U ## shift(9, 0)
      bitsReg := bitsReg - 1.U
    }.otherwise {
      when(io.channel.valid) {
        // two stop bits, data, one start bit
        shiftReg := 3.U ## io.channel.bits ## 0.U
        bitsReg := 11.U
      }.otherwise {
        shiftReg := 0x7ff.U
      }
    }

  }.otherwise {
    cntReg := cntReg - 1.U
  }
}
```

Listing 11.3: 직렬포트를위한송신기.

이터, 정지비트두개 (1) 로구성됩니다. 따라서비트개수는 11 로설정됩니다.

이매우최소한의송신기는추가버퍼가없으며, 시프트레지스터가비어있고 cntReg 가 0 인클록사이클에서만새로운문자를받아들일수있습니다. cntReg 가 0 일때만새데이터를받아들인다는것은, 시프트레지스터에공간이있을때에도 ready 플래그가비활성화됨을의미합니다. 그러나우리는이러한" 복잡성" 을송신기에추가하지않고, 이를버퍼에위임하고자합니다.

목록 11.4은버블 FIFO 의 FIFO 레지스터와유사한단일바이트버퍼를보여줍니다. 입력과출력은 UartIO 입니다. 이버퍼는 empty 또는 full 을나타내기위한최소한의상태기계를포함합니다. 버퍼가주도하는핸드셰이크신호 (io.in.ready 와 io.out.valid) 는상태레지스터에의존합니다.

상태가 empty 이고입력의데이터가 valid 일때, 우리는데이터를레지스터에저장하고 full 상태로전환합니다. 상태가 full 이고다운스트림수신자가 ready 일때, 다운스트림데이터전송이일어나고다시 empty 상태로전환합니다.

이버퍼를사용하면최소한의송신기를확장할수있습니다. 목록 11.5은송신기 Tx 와앞단의단일버퍼를결합한것을보여줍니다. 이버퍼는이제 Tx 가단한클록사이클동안만 ready 였던문제를완화합니다. 우리는이문제의해결을버퍼모듈에위임했습니다. 단일워드버퍼를실제 FIFO 로확장하는것은쉽게수행할수있으며, 송신기나단일바이트버퍼에어떠한변경도필요하지않습니다.

목록 11.6은수신기 (Rx) 의코드를보여줍니다. 수신기는직렬데이터의타이밍을재구성해야하므로다소까다롭습니다. 수신기는시작비트의하강에지를기다립니다. 그이벤트로부터, 수신기는비트 0 의중앙에위치하기위해 1.5 비트시간을기다립니다. 그런다음, 매비트시간마다샘플링하여비트를시프트인합니다. 이두대기시간은 BIT_CNT 와 START_CNT 로확인할수있습니다. 두샘플링시간모두동일한카운터 (cntReg) 가사용됩니다. 8 비트가시프트인되고나면, validReg 가바이트가사용가능함을신호합니다.

목록 11.7은친근한메시지를전송함으로써직렬포트송신기의사용법을보여줍니다. 우리는메시지를 Scala 문자열 (msg) 로정의하고, 이를 UInt 의 Chisel Vec 으로변환합니다. Scala 문자열은 map 메서드를지원하는시퀀스입니다. map 메서드는인자로함수리터럴을받아, 이함수를각원소에적용하고, 함수의반환값들로이루어진시퀀스를구성합니다. 이경우처럼함수리터럴이인자를하나만가진다면, 그인자는 _ 로표현할수있습니다. 우리의함수리터럴은 Chisel 메서드 .U 를호출하여 Scala Char 를 Chisel UInt 로변환합니다. 그런다음이시퀀스는 VecInit 에전달되어 Chisel Vec 을구성합니다. 우리는카운터 cntReg 로벡터 text 에인덱싱하여버퍼링된송신기에개별문자를제공합니다. 각 ready 신호가발생할때마다카운터를증가시켜전체문자열이전송될때까지반복합니다. 송신기 (sender) 는마지막문자가전송될때까지 valid 를계속활성화된상태로유지합니다.

목록 11.8은수신기와송신기를연결하여사용하는방법을보여줍니다. 이연결

```
class Buffer extends Module {
  val io = IO(new Bundle {
    val in = Flipped(new UartIO())
    val out = new UartIO()
  })

  object State extends ChiselEnum {
    val empty, full = Value
  }
  import State._

  val stateReg = RegInit(empty)
  val dataReg = RegInit(0.U(8.W))

  io.in.ready := stateReg === empty
  io.out.valid := stateReg === full

  when(stateReg === empty) {
    when(io.in.valid) {
      dataReg := io.in.bits
      stateReg := full
    }
  } .otherwise { // full
    when(io.out.ready) {
      stateReg := empty
    }
  }
  io.out.bits := dataReg
}
```

Listing 11.4: ready/valid 인터페이스를갖춘단일바이트버퍼.

```
class BufferedTx(frequency: Int, baudRate: Int) extends
    Module {
  val io = IO(new Bundle {
    val txd = Output(UInt(1.W))
    val channel = Flipped(new UartIO())
  })
  val tx = Module(new Tx(frequency, baudRate))
  val buf = Module(new Buffer())

  buf.io.in <> io.channel
  tx.io.channel <> buf.io.out
  io.txd <> tx.io.txd
}
```

Listing 11.5: 추가버퍼를갖춘송신기.

은수신된각문자가다시전송 (에코) 되는 Echo 회로를생성합니다.

11.3 FIFO 설계변형

이장의시작부분에서우리는단순한버블 FIFO 의설계를소개했습니다. 이절에서는 FIFO 를일반화하여큐의다양한변형을구현하겠습니다. 이러한구현들을서로교체가능하게만들기위해, 절 10.5에서소개한상속을사용하겠습니다.

11.3.1 FIFO 매개변수화하기

우리는임의의 Chisel 데이터타입을버퍼링할수있도록, Chisel 타입 T 를매개변수로갖는 제네릭클래스로서 abstract FIFO 클래스를정의합니다 (목록 11.9 참조). 이추상클래스에서는매개변수 depth 가유효한값을가지는지도검사합니다.

절 11.1에서우리는 write, full, din, read, empty, dout 과같이신호에대한일반적인이름을가진우리자신의인터페이스용타입을정의했습니다. 이러한버퍼의입력과출력은데이터와핸드셰이킹을위한두개의신호로구성됩니다 (예를들어, FIFO 가 full 이아닐때우리는 FIFO 에 write 합니다).

여기서는이러한핸드셰이킹을 ready/valid 인터페이스로일반화할수있습니다. FIFO 가 ready 일때원소를인큐 (FIFO 에쓰기) 할수있습니다. 우리는이를

```
class Rx(frequency: Int, baudRate: Int) extends Module {
  val io = IO(new Bundle {
    val rxd = Input(UInt(1.W))
    val channel = new UartIO()
  })

  val BIT_CNT = ((frequency + baudRate / 2) / baudRate - 1)
  val START_CNT = ((3 * frequency / 2 + baudRate / 2) /
      baudRate - 2) // -2 for the falling delay

  // Sync in the asynchronous RX data
  val rxReg = RegNext(RegNext(io.rxd, 0.U), 0.U)
  val falling = !rxReg && (RegNext(rxReg) === 1.U)

  val shiftReg = RegInit(0.U(8.W))
  val cntReg = RegInit(BIT_CNT.U(20.W)) // have some idle
      time before listening
  val bitsReg = RegInit(0.U(4.W))
  val valReg = RegInit(false.B)

  when(cntReg =/= 0.U) {
    cntReg := cntReg - 1.U
  }.elsewhen(bitsReg =/= 0.U) {
    cntReg := BIT_CNT.U
    shiftReg := Cat(rxReg, shiftReg >> 1)
    bitsReg := bitsReg - 1.U
    // the last shifted in
    when(bitsReg === 1.U) {
      valReg := true.B
    }
  }.elsewhen(falling) { // wait 1.5 bits after falling edge
      of start
    cntReg := START_CNT.U
    bitsReg := 8.U
  }

  when(valReg && io.channel.ready) {
    valReg := false.B
  }

  io.channel.bits := shiftReg
  io.channel.valid := valReg
}
```

```
class Sender(frequency: Int, baudRate: Int) extends Module {
  val io = IO(new Bundle {
    val txd = Output(UInt(1.W))
  })

  val tx = Module(new BufferedTx(frequency, baudRate))

  io.txd := tx.io.txd

  val msg = "Hello World!"
  val text = VecInit(msg.map(_.U))
  val len = msg.length.U

  val cntReg = RegInit(0.U(8.W))

  tx.io.channel.bits := text(cntReg)
  tx.io.channel.valid := cntReg =/= len

  when(tx.io.channel.ready && cntReg =/= len) {
    cntReg := cntReg + 1.U
  }
}
```

Listing 11.7: 직렬포트를통해"Hello World!" 를전송하기.

```
class Echo(frequency: Int, baudRate: Int) extends Module {
  val io = IO(new Bundle {
    val txd = Output(UInt(1.W))
    val rxd = Input(UInt(1.W))
  })
  val tx = Module(new BufferedTx(frequency, baudRate))
  val rx = Module(new Rx(frequency, baudRate))
  io.txd := tx.io.txd
  rx.io.rxd := io.rxd
  tx.io.channel <> rx.io.channel
}
```

Listing 11.8: 직렬포트에서데이터를에코하기.

```
abstract class Fifo[T <: Data](gen: T, val depth: Int)
    extends Module {
  val io = IO(new FifoIO(gen))

  require(depth > 0, "Number of buffer elements needs to be
      larger than 0")
}
```

Listing 11.9: FIFO 변형을위한추상클래스.

쓰기측에서 valid 로신호합니다. 이 ready/valid 인터페이스는매우흔하므로, Chisel 은이인터페이스의정의를다음과같이 DecoupledIO 로제공합니다:[2]

```
class DecoupledIO[T <: Data](gen: T) extends Bundle {
  val ready = Input(Bool())
  val valid = Output(Bool())
  val bits  = Output(gen)
}
```

DecoupledIO 인터페이스를사용하여 FIFO 의인터페이스를정의합니다: 인큐 (enq) 와디큐 (deq) 포트로구성되며 read/valid 인터페이스로이루어진 FifoIO 입니다. DecoupledIO 인터페이스는작성자 (생산자) 의관점에서정의됩니다. 따라서 FIFO 의인큐포트는신호방향을뒤집어야합니다.

```
class FifoIO[T <: Data](private val gen: T) extends Bundle {
  val enq = Flipped(new DecoupledIO(gen))
  val deq = new DecoupledIO(gen)
}
```

추상기본클래스와인터페이스를사용하면서로다른매개변수 (속도, 면적, 전력, 또는단순함) 에최적화된다양한 FIFO 구현을특화할수있습니다.

11.3.2 버블 FIFO 재설계

Section 11.1의버블 FIFO 를표준 ready/valid 인터페이스를사용하고 Chisel 데이터타입으로매개변수화할수있도록재정의할수있습니다.

[2] 이는단순화된것으로, DecoupledIO 는실제로는추상클래스를확장합니다.

```
class BubbleFifo[T <: Data](gen: T, depth: Int) extends
    Fifo(gen: T, depth: Int) {

  private class Buffer() extends Module {
    val io = IO(new FifoIO(gen))

    val fullReg = RegInit(false.B)
    val dataReg = Reg(gen)

    when(fullReg) {
      when(io.deq.ready) {
        fullReg := false.B
      }
    }.otherwise {
      when(io.enq.valid) {
        fullReg := true.B
        dataReg := io.enq.bits
      }
    }

    io.enq.ready := !fullReg
    io.deq.valid := fullReg
    io.deq.bits := dataReg
  }

  private val buffers = Array.fill(depth) { Module(new
      Buffer()) }
  for (i <- 0 until depth - 1) {
    buffers(i + 1).io.enq <> buffers(i).io.deq
  }

  io.enq <> buffers(0).io.enq
  io.deq <> buffers(depth - 1).io.deq
}
```

Listing 11.10: ready/valid 인터페이스를갖춘버블 FIFO.

Listing 11.10은 ready/valid 인터페이스를갖도록리팩터링된버블 FIFO를보여줍니다. Buffer 컴포넌트를 BubbleFifo 내부에비공개클래스로넣었다는점에주목하십시오. 이헬퍼클래스는이컴포넌트에만필요하므로숨겨서네임스페이스가오염되지않도록합니다. 버퍼클래스도단순화되었습니다. FSM 대신, 버

퍼의상태 (가득참또는비어있음) 를나타내기위해단일비트 (fullReg) 만사용합니다.

버블 FIFO 는단순하고이해하기쉬우며최소한의자원을사용합니다. 그러나각버퍼단계가비어있음과가득참사이를전환해야하므로, 이 FIFO 의최대대역폭은두클록사이클마다한워드입니다.

버퍼에서양쪽인터페이스를모두살펴봄으로써생산자의 valid 와소비자의 ready 가동시에만족될때새워드를받아들일수있게하는방법도고려해볼수있습니다. 그러나이는소비자핸드셰이크에서생산자핸드셰이크로이어지는조합경로를도입하게되어, ready/valid 프로토콜의의미를위반합니다.

11.3.3 이중버퍼 FIFO

한가지해결책은버퍼레지스터가가득찬상태에서도 ready 를유지하는것입니다. 소비자가 ready 가아닐때생산자로부터데이터워드를받아들이려면두번째버퍼가필요한데, 이를섀도레지스터라고부릅니다. 버퍼가가득찬경우, 새데이터는섀도레지스터에저장되고 ready 는비활성화됩니다. 소비자가다시 ready 가되면, 데이터는데이터레지스터에서소비자로, 그리고섀도레지스터에서데이터레지스터로전달됩니다.

```
class DoubleBufferFifo[T <: Data](gen: T, depth: Int)
    extends Fifo(gen: T, depth: Int) {

  private class DoubleBuffer[T <: Data](gen: T) extends
      Module {
    val io = IO(new FifoIO(gen))

    object State extends ChiselEnum {
      val empty, one, two = Value
    }
    import State._

    val stateReg = RegInit(empty)
    val dataReg = Reg(gen)
    val shadowReg = Reg(gen)

    switch(stateReg) {
      is(empty) {
        when(io.enq.valid) {
          stateReg := one
          dataReg := io.enq.bits
        }
      }
      is(one) {
        when(io.deq.ready && !io.enq.valid) {
          stateReg := empty
        }
        when(io.deq.ready && io.enq.valid) {
          stateReg := one
          dataReg := io.enq.bits
        }
        when(!io.deq.ready && io.enq.valid) {
          stateReg := two
          shadowReg := io.enq.bits
        }
      }
      is(two) {
        when(io.deq.ready) {
          dataReg := shadowReg
          stateReg := one
```

```
        }
      }
    }

    io.enq.ready := (stateReg === empty || stateReg === one)
    io.deq.valid := (stateReg === one || stateReg === two)
    io.deq.bits := dataReg
  }

  private val buffers = Array.fill((depth + 1) / 2) {
      Module(new DoubleBuffer(gen)) }

  for (i <- 0 until (depth + 1) / 2 - 1) {
    buffers(i + 1).io.enq <> buffers(i).io.deq
  }
  io.enq <> buffers(0).io.enq
  io.deq <> buffers((depth + 1) / 2 - 1).io.deq
}
```

Listing 11.11: 이중버퍼요소를갖춘 FIFO.

목록 11.11은이중버퍼 FIFO 를보여줍니다. 각버퍼요소가두개의항목을저장할수있으므로, 버퍼요소의절반 (depth/2) 만필요합니다. DoubleBuffer는 dataReg 와 shadowReg 라는두개의레지스터를포함합니다. 소비자는항상 dataReg 로부터서비스를받습니다. 이중버퍼는 empty, one, two 라는세가지상태를가지며, 이는이중버퍼의충전수준을나타내는신호입니다. 버퍼는상태가 empty 또는 one 일때새데이터를받을 ready 상태가됩니다. 버퍼는상태가 one 또는 two 일때유효한데이터를가집니다.

FIFO 를전속력으로실행하고소비자가항상 ready 상태라면, 더블버퍼의정상상태는 one 입니다. 소비자가 ready 를비활성화할때만큐가채워지고버퍼는 two 상태로진입합니다. 그러나단일버블 FIFO 와비교하면, 동일한버퍼용량에대해큐의재시작은클록사이클수의절반만소요됩니다. 마찬가지로폴스루지연시간도버블 FIFO 의절반입니다.

11.3.4 레지스터메모리를사용한 FIFO

소프트웨어공학배경을가지고있다면, 왜우리가병렬로실행되며상류및하류요소와핸드셰이크하는여러개의작은버퍼요소로하드웨어큐를구성했는지의아했을수있습니다. 작은버퍼의경우, 이것이아마도가장효율적인구현일것입니다.

소프트웨어에서큐는보통두스레드에서순차적코드에의해사용됩니다. 생산자스레드와소비자스레드를분리하기위해큐를사용합니다. 이러한설정에서, 고정크기 FIFO 큐는보통 원형버퍼로구현됩니다. 두개의포인터가큐를위해마련된메모리내의읽기위치와쓰기위치를가리킵니다. 포인터가메모리의끝에도달하면해당메모리의시작지점으로되돌아갑니다. 두포인터사이의차이가큐안의요소개수입니다. 두포인터가같은주소를가리키면큐는비어있거나가득찬상태입니다. 비어있음과가득참을구별하려면또다른플래그가필요합니다.

이러한메모리기반 FIFO 큐를하드웨어로도구현할수있습니다. 작은큐의경우, 레지스터파일 (즉, Reg(Vec())) 을사용할수있습니다. Listing 11.12은메모리와읽기·쓰기포인터로구현된 FIFO 큐를보여줍니다.

```
class RegFifo[T <: Data](gen: T, depth: Int) extends
    Fifo(gen: T, depth: Int) {

  def counter(depth: Int, incr: Bool): (UInt, UInt) = {
    val cntReg = RegInit(0.U(log2Ceil(depth).W))
    val nextVal = Mux(cntReg === (depth - 1).U, 0.U, cntReg
        + 1.U)
    when(incr) {
      cntReg := nextVal
    }
    (cntReg, nextVal)
  }

  // the register based memory
  val memReg = Reg(Vec(depth, gen))

  val incrRead = WireDefault(false.B)
  val incrWrite = WireDefault(false.B)
  val (readPtr, nextRead) = counter(depth, incrRead)
  val (writePtr, nextWrite) = counter(depth, incrWrite)

  val emptyReg = RegInit(true.B)
  val fullReg = RegInit(false.B)

  val op = io.enq.valid ## io.deq.ready
  val doWrite = WireDefault(false.B)

  switch(op) {
    is("b00".U) {}
    is("b01".U) { // read
      when(!emptyReg) {
        fullReg := false.B
        emptyReg := nextRead === writePtr
        incrRead := true.B
      }
    }
    is("b10".U) { // write
      when(!fullReg) {
        doWrite := true.B
        emptyReg := false.B
```

```
        fullReg := nextWrite === readPtr
        incrWrite := true.B
      }
    }
    is("b11".U) { // write and read
      when(!fullReg) {
        doWrite := true.B
        emptyReg := false.B
        when(emptyReg) {
          fullReg := false.B
        }.otherwise {
          fullReg := nextWrite === nextRead
        }
        incrWrite := true.B
      }
      when(!emptyReg) {
        fullReg := false.B
        when(fullReg) {
          emptyReg := false.B
        }.otherwise {
          emptyReg := nextRead === nextWrite
        }
        incrRead := true.B
      }
    }
  }

  when(doWrite) {
    memReg(writePtr) := io.enq.bits
  }

  io.deq.bits := memReg(readPtr)
  io.enq.ready := !fullReg
  io.deq.valid := !emptyReg
}
```

Listing 11.12: 레지스터기반메모리를갖춘 FIFO.

동작이있을때마다증가하고버퍼의끝에서랩어라운드되는두개의포인터가 있으므로, 이러한랩어라운드카운터를구현하는함수 counter() 를정의합니다. log2Ceil(depth).W 를사용하여카운터의비트길이를계산합니다. 다음값은 1 증

가하거나 0 으로랩어라운드되는것중하나입니다. 카운터는입력 incr 이 true.B 일때만증가합니다.

더나아가, (증가또는래핑시 0 으로되는) 가능한다음값도필요하므로, 이값을 counter 함수에서도반환합니다. Scala 에서는 튜플을반환할수있는데, 이는단순히하나이상의값을담는컨테이너입니다. 튜플을생성하는문법은쉼표로구분된값들을괄호로감싸기만하면됩니다:

```
val t = (v1, v2)
```

대입문의좌변에서괄호표기법을사용하여이러한튜플을분해할수있습니다.

```
val (x1, x2) = t
```

메모리로는 Chisel 데이터타입 gen 의벡터레지스터 (Reg(Vec(depth, gen))) 를사용합니다. 읽기포인터와쓰기포인터를증가시키기위한두신호를정의하고, 함수 counter 를사용하여읽기포인터와쓰기포인터를생성합니다. 두포인터가같으면버퍼는비어있거나가득찬상태입니다. 비어있음과가득참을나타내기위한두개의플래그를정의합니다.

생산자가 valid 를표명하고 FIFO 가가득차지않은경우, 우리는다음을수행합니다: (1) 버퍼에씁니다, (2) emptyReg 가비표명상태임을보장합니다, (3) 다음클록사이클에쓰기포인터가읽기포인터를따라잡게되면 (현재읽기포인터와다음쓰기포인터를비교) 버퍼를가득찬것으로표시합니다, (4) 쓰기카운터를증가시키도록신호를보냅니다.

소비자가 ready 상태이고 FIFO 가비어있지않은경우, 우리는다음을수행합니다: (1) fullReg 가비표명상태임을보장합니다, (2) 다음클록사이클에읽기포인터가쓰기포인터를따라잡게되면버퍼를비어있는것으로표시합니다, (3) 읽기카운터를증가시키도록신호를보냅니다.

읽기와쓰기를동시에수행하는것도가능합니다. 이경우는앞의두경우를종합한것입니다. *TODO:* 경계사례가올바른지확신할수없습니다. 가득찬버퍼에대한읽기와쓰기도정상적으로동작해야하지않을까요*?*

FIFO 의출력은읽기포인터주소에있는메모리요소입니다. ready 및 valid 플래그는단순히 full 및 empty 플래그로부터유도됩니다.

11.3.5 온칩메모리를사용하는 FIFO

앞선버전의 FIFO 는메모리를표현하기위해레지스터파일을사용했으며, 이는작은 FIFO 에적합한해법입니다. 더큰 FIFO 의경우, 온칩메모리를사용하는편이더좋습니다. Listing 11.13은저장소로동기식메모리를사용하는 FIFO 를보여줍니다.

```
class MemFifo[T <: Data](gen: T, depth: Int) extends
    Fifo(gen: T, depth: Int) {

  def counter(depth: Int, incr: Bool): (UInt, UInt) = {
    val cntReg = RegInit(0.U(log2Ceil(depth).W))
    val nextVal = Mux(cntReg === (depth - 1).U, 0.U, cntReg
        + 1.U)
    when(incr) {
      cntReg := nextVal
    }
    (cntReg, nextVal)
  }

  val mem = SyncReadMem(depth, gen, SyncReadMem.WriteFirst)

  val incrRead = WireDefault(false.B)
  val incrWrite = WireDefault(false.B)
  val (readPtr, nextRead) = counter(depth, incrRead)
  val (writePtr, nextWrite) = counter(depth, incrWrite)

  val emptyReg = RegInit(true.B)
  val fullReg = RegInit(false.B)

  val outputReg = Reg(gen)
  val outputValidReg = RegInit(false.B)
  val read = WireDefault(false.B)

  io.deq.valid := outputValidReg
  io.enq.ready := !fullReg

  val doWrite = WireDefault(false.B)
  val data = Wire(gen)
  data := mem.read(readPtr)
  io.deq.bits := data
  when(doWrite) {
    mem.write(writePtr, io.enq.bits)
  }

  val readCond =
    !outputValidReg && ((readPtr =/= writePtr) || fullReg)
```

```
        // should add optimization when downstream is ready
        for pipielining
  when(readCond) {
    read := true.B
    incrRead := true.B
    outputReg := data
    outputValidReg := true.B
    emptyReg := nextRead === writePtr
    fullReg := false.B // no concurrent read when full (at
        the moment)
  }
  when(io.deq.fire) {
    outputValidReg := false.B
  }
  io.deq.bits := outputReg

  when(io.enq.fire) {
    emptyReg := false.B
    fullReg := (nextWrite === readPtr) & !read
    incrWrite := true.B
    doWrite := true.B
  }
}
```

Listing 11.13: 온칩메모리를사용하는 FIFO.

읽기및쓰기포인터의처리방식은레지스터메모리 FIFO 와동일합니다. 다만, 동기식온칩메모리는다음클록사이클에읽기결과를전달하는반면, 레지스터파일의읽기는같은클록사이클내에사용가능했습니다. 따라서이지연시간을처리하기위해추가레지스터가필요합니다.

TODO: 아래 *(*주석에*)* 에설명된대로 *FIFO* 를다시구현해야할수도있습니다.

TODO: 메모리기반 *FIFO* 는대량의데이터를큐에효율적으로보관할수있으며통과지연시간이짧습니다. 앞선설계에서, *FIFO* 의출력은메모리읽기로부터직접나올수있습니다. 이데이터패스가설계의임계경로에있는경우, 두개의 *FIFO* 를결합함으로써손쉽게설계를파이프라인화할수있습니다. *Listing 11.14*는이러한결합을보여줍니다. 메모리기반 *FIFO* 의출력단에단일단계이중버퍼 *FIFO* 를추가하여메모리읽기경로를출력으로부터분리합니다.

```
class CombFifo[T <: Data](gen: T, depth: Int) extends
    Fifo(gen: T, depth: Int) {
```

```
  val memFifo = Module(new MemFifo(gen, depth))
  val bufferFIFO = Module(new DoubleBufferFifo(gen, 2))
  io.enq <> memFifo.io.enq
  memFifo.io.deq <> bufferFIFO.io.enq
  bufferFIFO.io.deq <> io.deq
}
```

Listing 11.14: 메모리기반 FIFO 와이중버퍼단계를결합.

11.4 다중클록메모리

여러클록도메인을가진대규모설계에서는, 한도메인에서다른도메인으로데이터를안전하게전달할방법이필요할수있습니다. 앞서이문제에대한한가지해법으로동기화를살펴본바있습니다. 또다른대안은두 (또는그이상의) 도메인사이의버퍼로다중클록메모리를사용하는것입니다.

Chisel 은 withClock 및 withClockAndReset 구문을통해다중클록설계를지원합니다. withClock(clk) 블록내에서정의된모든저장요소는 clk 에의해클록킹됩니다. 다중클록메모리의경우, 메모리모듈은모든 withClock 블록바깥에서정의되어야하며, 각포트는자신만의 withClock 블록을가져야합니다. 매개변수화된다중클록메모리가 Listing 11.15에나와있습니다.

```
class MemoryIO(val n: Int, val w: Int) extends Bundle {
  val clk = Input(Bool())
  val addr  = Input(UInt(log2Up(n).W))
  val datai = Input(UInt(w.W))
  val datao = Output(UInt(w.W))
  val en = Input(Bool())
  val we = Input(Bool())
}

class MultiClockMemory(ports: Int, n: Int = 1024, w: Int =
    32) extends Module {
  val io = IO(new Bundle {
    val ps = Vec(ports, new MemoryIO(n, w))
  })

  val ram = SyncReadMem(n, UInt(w.W))
```

```
  /* does not work in Chisel 3.5.6
  for (i <- 0 until ports) {
    val p = io.ps(i)
    p.datao := ram.readWrite(p.addr, p.datai, p.en, p.we,
        p.clk.asClock)
  }
   */
}
```

Listing 11.15: 다중클록메모리생성기.

당연히, 이러한다중클록메모리를사용하면동시에수행할수있는연산에일부 제약이생깁니다. 두 (또는그이상의) 포트가같은주소에동시에쓸수없는데, 이는준안정성을유발할수있기때문입니다. 마찬가지로, 원하는 write-during-read(쓰기중읽기) 동작을정의해두어야합니다. 메모리는입력데이터가읽기포트로전달되는 write-first(쓰기우선) 방식이나, 이전메모리값이읽기포트에제시되는 read-first(읽기우선) 방식중하나로구성되어야합니다.

다만, ChiselTest 의다중클록지원은아직매우초기단계에있다는점에유의하십시오. 전이를강제하려면클록신호를수동으로토글해야합니다.

11.5 연습문제

이연습문제절은두가지연습을포함하고있어다소깁니다: (1) 버블 FIFO 를탐구하고다른 FIFO 설계를구현하는것, (2) UART 를탐구하고이를확장하는것입니다. 두연습문제의소스코드는모두 chisel-examples 저장소에포함되어있습니다.

11.5.1 버블 FIFO 탐구하기

FIFO 소스에는다양한읽기및쓰기동작을유발하고 value change dump (VCD) 형식으로파형을생성하는테스터도포함되어있습니다. VCD 파일은 GTKWave와같은파형뷰어로볼수있습니다. 저장소에있는 FifoSpec을탐구해보십시오. 저장소에는예제를실행하기위한 Makefile 이포함되어있으며, FIFO 예제의경우다음과같이입력하기만하면됩니다:

```
$ make fifo
```

이 make 명령은 FIFO 를컴파일하고, 테스트를실행한뒤, 파형확인을위해 GTKWave 를시작합니다.[3] 테스터와생성된파형을탐구해보십시오.

첫번째사이클에서, 테스터는단일워드를씁니다. 파형에서이워드가 FIFO 를통해어떻게버블 (bubble) 처럼이동하는지관찰할수있으며, 이때문에 버블 *FIFO* 라는이름이붙었습니다. 이러한버블링은또한 FIFO 를통과하는데이터 워드의지연시간이 FIFO 의깊이와같음을의미합니다.

다음테스트는 FIFO 가가득찰때까지채웁니다. 이어서단일읽기가수행됩니다. 빈워드가 FIFO 의읽기측에서쓰기측으로어떻게버블링되는지주목하십시오. 버블 FIFO 가가득찬경우, 읽기가쓰기측에영향을미치려면버퍼깊이만큼의 지연시간이소요됩니다.

테스트의마지막부분에는최대속도로쓰기와읽기를시도하는루프가포함되어 있습니다. 버블 FIFO 가워드당두클록사이클이라는최대대역폭으로동작하는 것을확인할수있습니다. 단일워드전송의경우버퍼단계는항상 empty 와 full 사이를오가야합니다.

버블 FIFO 는단순하며, 작은버퍼의경우자원요구량이적습니다. n 단계버블 FIFO 의주요단점은다음과같습니다: (1) 최대처리량이두클록사이클당한워드라는점, (2) 데이터워드가쓰기측끝에서읽기측끝까지이동하는데 n 클록사이클이걸린다는점, (3) FIFO 가가득찬경우재시작에 n 클록사이클이필요하다는점입니다.

이러한단점들은 순환버퍼 (circular buffer)를사용한 FIFO 구현으로해결할수있습니다. 순환버퍼는메모리와읽기·쓰기포인터로구현할수있습니다. 다른 FIFO 구현으로테스트를다시실행하거나다시작성하여대역폭과지연시간을 비교해보십시오. 서로다른버전의 FIFO 를합성하여자원요구량을비교해보십시오.

11.5.2 UART

UART 예제를위해서는직렬포트가있는 FPGA 보드와노트북용직렬포트 (일반적으로 USB 연결) 가필요합니다. FPGA 보드와노트북의직렬포트사이를직렬 케이블로연결하십시오. 터미널프로그램, 예를들어 Windows 의 Hyperterm 이나 Linux 의 gtkterm 을실행하십시오:

```
$ gtkterm &
```

올바른장치를사용하도록포트를설정하십시오. USB UART 의경우흔히 /dev/ttyUSB0 과같은형태입니다. 보드율을 115200 으로설정하고패리티나흐

[3] 운영체제에따라 GKTWave 를수동으로시작해야할수도있습니다

름제어 (핸드셰이크) 는사용하지않도록설정하십시오. 다음명령으로 UART 용 Verilog 코드를생성할수있습니다:

```
$ make uart
```

그런다음합성도구를사용하여설계를합성하십시오. 리포지토리에는 DE2-115 FPGA 보드용 Quartus 프로젝트가포함되어있습니다. Quartus 에서는재생 버튼을사용하여설계를합성하고 FPGA 를구성하십시오. 구성이끝나면터미널 에인사말메시지가표시되어야합니다.

깜빡이는 LED 예제를 UART 로확장하여, LED 가꺼져있을때와켜져있을 때직렬라인에각각 0 과 1 을기록하도록하십시오. Sender 예제에서와같이 BufferedTx 를사용하십시오.

문자출력속도가느린경우 (초당두개), UART 송신레지스터에데이터를기록 하면서 ready/valid 핸드셰이크를무시할수있습니다. 예제를확장하여보드율 이허용하는최대속도로 0-9 까지의숫자를반복해서기록하도록해보십시오. 이 경우에는송신버퍼가비어있는지확인하기위해 UART 상태를폴링하도록상태기 계를확장해야합니다.

예제코드에는 Tx 용버퍼가하나만포함되어있습니다. 여러분이구현한 FIFO 를추가하여송신기와수신기에버퍼링을자유롭게추가해보십시오.

11.5.3 FIFO 탐구

전용레지스터에네개의버퍼요소를갖는단순한 FIFO 를작성해보십시오. 오버 플로가그대로발생할수있는 2 비트읽기·쓰기카운터를사용하십시오. 추가적인 단순화로서, 읽기포인터와쓰기포인터가같을때를빈 FIFO 로간주하는상황을 고려해보십시오. 이는최대 3 개의요소만저장할수있음을의미합니다. 이러한 단순화를통해 Listing 11.12의예제에있던카운터함수와, 동일한포인터값으로 empty 와 full 을처리하는문제를피할수있습니다. empty 나 full 플래그는포 인터값만으로유도할수있으므로필요하지않습니다. 이설계는얼마나더단순해졌 습니까?

지금까지제시한서로다른 FIFO 설계들은다음속성들에대해서로다른설계절 충을가집니다: (1) 최대처리량, (2) 폴스루지연시간, (3) 자원요구량, (4) 최 대클록주파수. FPGA 용으로합성하여다양한크기의모든 FIFO 변형을탐구해 보십시오. 소스는 ip-contributions에서확인할수있습니다. 4 워드, 16 워드, 256 워드 FIFO 의최적점은어디입니까?

12 인터커넥트

우리는서로다른컴포넌트들을결합하여더큰시스템을구축합니다. 컴포넌트의조합을단순화하기위해 Wishbone이나 AXI 같은인터커넥트표준이존재합니다. 이장에서는다양한형태의인터커넥트를살펴봅니다.

인터커넥트는칩간 (외부) 에서사용될수도있고, 칩내부에서사용될수도있는데, 후자의경우흔히시스템-온-칩 (SoC) 이라고부릅니다.

12.1 고전적인마이크로프로세서버스

Figure 12.1는단순하고고전적인컴퓨터의개략도를보여줍니다. 중앙처리장치 (CPU) 는 시스템버스를통해외부메모리및입출력 (I/O) 장치에연결됩니다. 이러한형태의버스인터커넥션은 Z80이나 6502 같은초기마이크로프로세서에서흔히볼수있었습니다.

버스는주소버스, 데이터버스, 그리고 읽기와 쓰기같은제어신호로나뉩니다. CPU 는주소신호와제어신호를구동합니다. 데이터버스는양방향입니다. CPU 는시스템에서마스터역할을하며, 읽기또는쓰기명령을내립니다. 두명령모두메모리에서데이터워드를선택하거나 I/O 장치의레지스터를선택하기위한주소 (개략도의 addr) 를포함합니다. 모든주소라인이모든주변장치에연결되어있는것은아닙니다. 서로다른장치를선택하기위해주소버스의상위비트들이디코더의입력으로사용됩니다. 디코더의출력은주변장치들의칩선택 (CS) 입력에연결됩니다.

읽기명령이내려지면, 선택된장치는일정한접근시간이지난후데이터버스에데이터를제공합니다. 이때주변장치가데이터버스를구동합니다. 쓰기명령이내려지면, 프로세서가데이터를제공하고주변장치는그데이터를받아들여야합니다 (흔히신호의상승에지에서). 이때는 CPU 가데이터버스를구동합니다. 데이터버스는양방향이며데이터라인이모든장치들사이에서공유되므로, 출력에는 트라이스테이트 (tri-state) 드라이버가포함되어야합니다. 트라이스테이트구성에서는두출력트랜지스터가모두비활성화되어, 출력핀이사실상논리회로와분리됩니다.

가장단순한형태에서는버스에클록이없다는점에유의하십시오. 타이밍은주변장치들의읽기및쓰기접근시간에의해결정됩니다.

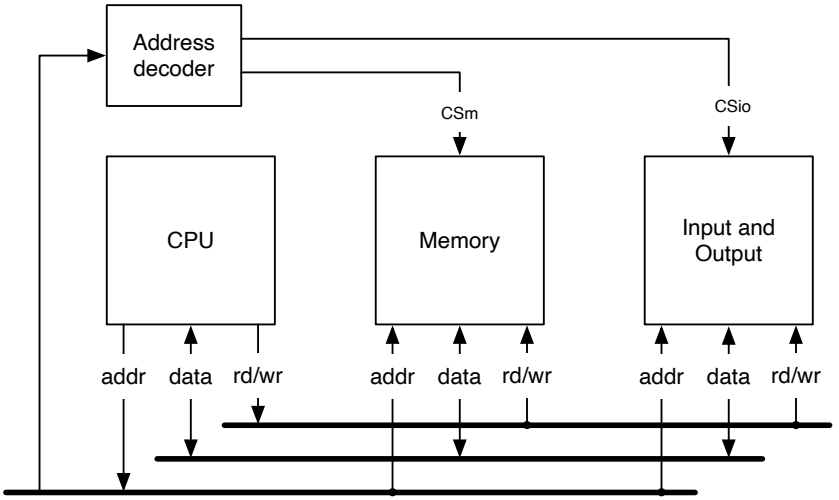


Figure 12.1: 프로세서 (CPU), 메모리, I/O 로구성되며, 주소버스, 데이터버스, 제어버스를통해연결되는고전적인컴퓨터.

최신컴퓨터는주변장치마다서로다른버스를사용합니다. 예를들어, 외부메모리를위한전용메모리버스와주변장치를위한 I/O 버스가그것입니다. 더욱이, PCI Express와같은최신 I/O 버스는직렬버스이며점대점연결을사용합니다.

그럼에도불구하고, 주소버스, 데이터버스, 칩선택신호로구성된고전적인프로세서버스는여전히코어상호연결에대한주류사고방식입니다. 다음절에서는이개념을온칩상호연결에맞게적용해보겠습니다.

12.2 온칩버스

이러한외부버스의개념을온칩버스로변환할수있습니다. 그러나몇가지측면을조정해야합니다. 트라이스테이트드라이버를요구하는공유버스는칩내부에서는실용적이지않습니다. 더욱이, 칩내부의배선은 PCB 나커넥터의배선보다저렴합니다. 따라서데이터버스를읽기신호용과쓰기신호용, 두개의배선집합으로분리합니다. 또한, 온칩연결은타이밍을정의하기위해클록을사용합니다.

그림 12.2은칩내부에서버스개념을구현한모습을보여줍니다. 주소, 데이터 출력, 제어신호는 CPU 에서모든주변장치로연결됩니다. 데이터입력에는 (트

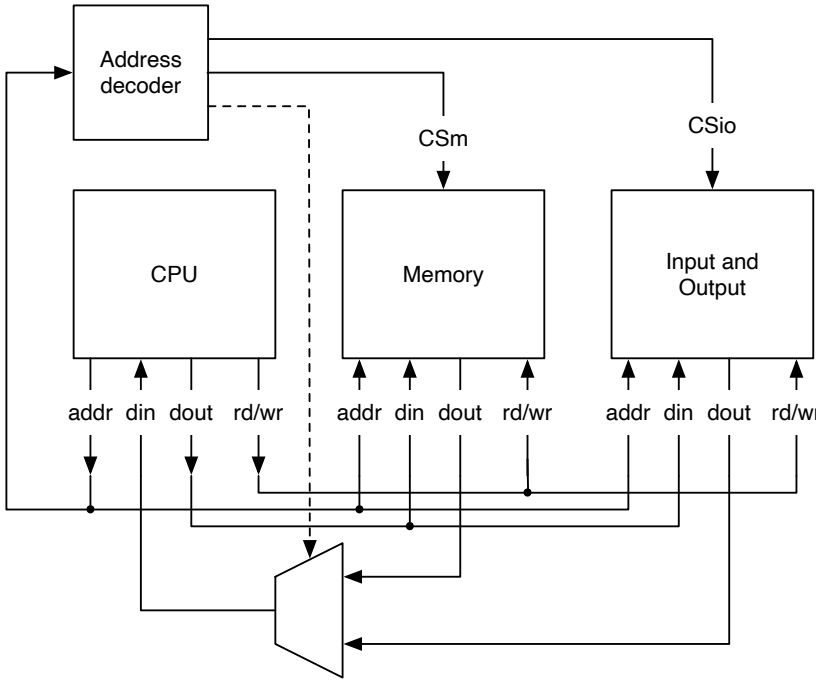


Figure 12.2: 오프칩버스개념을온칩” 버스” 로변환한것입니다.

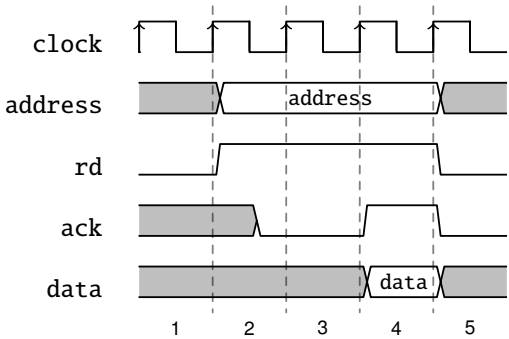


Figure 12.3: 조합적확인신호를사용하는읽기트랜잭션입니다.

라이스테이트버스대신) 멀티플렉서를사용합니다. 주소디코더는칩선택신호를생성하는것외에도, 데이터입력멀티플렉서의선택을제어합니다.

이단순한구성에서는각연산 (읽기또는쓰기) 이단일클록사이클내에실행될수있다고가정합니다. 이는매우작은시스템에서만가능합니다. 읽기요청다음클록사이클에읽기결과를기대하는것으로정의를확장할수있습니다. 이는한클록사이클의지연시간을갖는동기식읽기를사용하는온칩메모리에잘맞습니다. IO 장치의경우에도이추가적인클록사이클지연시간은타이밍제약을완화해줍니다. 여전히쓰기는한클록사이클내에수행된다고가정합니다.

지연시간이다르거나가변적인장치와통신하고자한다면, 핸드셰이크를도입해야합니다. 프로세서는읽기또는쓰기요청으로트랜잭션의시작을알리고, 메모리나주변장치는확인신호로트랜잭션의종료를알립니다.

12.2.1 조합핸드셰이크

그림 12.3는확인신호를사용한읽기요청을보여줍니다. 프로세서는클록사이클 2 에서주소버스 (address) 와읽기신호 (rd) 를구동합니다. 신호 ack 는그첫번째클록사이클내에반응해야합니다. 본예제에서읽기데이터는한클록사이클내에는사용할수없으며, 클록사이클 4 에서볼수있듯이두클록사이클후에야사용할수있습니다. 데이터와확인신호는단일클록사이클동안만유효합니다. 이프로토콜명세의장점은단일사이클트랜잭션을허용한다는것입니다. 그러나그대가로, 디코딩을포함한핸드셰이크과정이조합회로가되어, 최대주파수와관련된문제를일으킬수있습니다. 표준 Wishbone [16] 프로토콜은동일사이클확인을사용합니다. Wishbone 의최신버전에는파이프라인프로토콜이추가되었습니다.

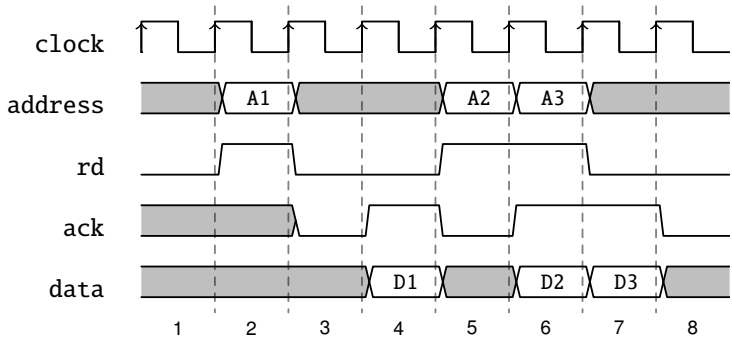


Figure 12.4: 파이프라인확인신호를사용하는읽기트랜잭션입니다.

동일사이클확인 (또는준비신호) 은 [17] 에서비판받은바있습니다. 단일사이클트랜잭션은대개더큰시스템에서는현실적이지않습니다. 따라서확인 (또는사용중이거나준비됨) 신호가요청사이클내에유효할필요가없는명세를정의할수있습니다. 해당논문은프로세서, 주소디코딩, 주변장치사이의조합경로를피하면서파이프라인트랜잭션을가능하게하는프로토콜인 SimpCon 을제안합니다.

12.2.2 파이프라인핸드셰이크

TODO: 다른것에상대적으로만서술하지마십시오. 그자체로독립적인것으로서술하십시오. 그런다음비교해도좋습니다. 이부분은좀더나은글쓰기가필요하며, *soc-comm README* 에도넣어야합니다. 또한쓰기타이밍다이어그램도있어야합니다. 그리고 *SVG* 에서더보기좋게렌더링되도록다이어그램에더나은글꼴을찾아야합니다.

여기서는단일사이클조합루프를피하고최신 SoC 설계에더적합한단순한파이프라인핸드셰이크버스프로토콜을정의합니다. 읽기또는쓰기명령은 rd 또는 wr 을단일클록사이클동안활성화함으로써신호를보냅니다. 주소와 (쓰기인경우) 쓰기데이터는명령이유효한동안함께유효해야합니다. 명령은단일사이클동안만유효합니다. 각명령은명령이있은후최소한사이클이지난시점에활성화된 ack 로확인되어야합니다. ack 를지연시켜대기상태를삽입할수도있습니다. 읽기데이터는 ack 신호와함께한클록사이클동안사용할수있습니다.

그림 12.4는주변장치의조합적반응이필요없는버스프로토콜을보여줍니다. 프로세서로부터의요청은단한클록사이클만지속됩니다. 주소버스와읽기신호는확인신호가있을때까지계속구동할필요가없습니다. 이전프로토콜과비교하면,

ack 신호는클록사이클 3 에서의 rd 명령이후최소한클록사이클이지나야유효 (로우또는하이) 할수있습니다. 이예제에서첫번째읽기시퀀스는두클록사이클의 지연시간을갖습니다. 이는이전예제와동일한지연시간입니다. 그러나요청이단 한클록사이클동안만유효하면되므로, 요청을파이프라인화할수있습니다. 주소 A2 와 A3 의읽기는연이어요청될수있어, 클록사이클당데이터워드 1 개의처리 량을허용합니다.

이상호연결을 PipeCon 이라고부르며, 이는인터페이스의파이프라인적특성을 강조하기위함입니다. 인터페이스는다음과같이정의됩니다.

```
class PipeCon(private val addrWidth: Int) extends Bundle {
  val address = Input(UInt(addrWidth.W))
  val rd = Input(Bool())
  val wr = Input(Bool())
  val rdData = Output(UInt(32.W))
  val wrData = Input(UInt(32.W))
  val wrMask = Input(UInt(4.W))
  val ack = Output(Bool())
}
```

Patmos 프로세서 [27] 는 IO 장치에접근하는데정확히이프로토콜을사용 하는 OCP 버전을사용합니다. 메모리는버스트인터페이스를통해연결됩니다. Patmos 핸드북 [22] 은사용된 OCP 인터페이스에대한상세한설명을제공합니 다. 더욱이, 여기서설명한파이프라인인터페이스를구현하는네트워크-온-칩과 같은멀티코어장치를담은 Chisel 저장소를시작하였습니다.

상호연결정의의온칩버전은점대점연결로일반화될수있습니다. 프로세서와주 변장치는이러한점대점인터페이스를통해스위칭패브릭에연결됩니다. 시스템에 하나이상의프로세서 (또는마스터) 가포함된경우, 어느마스터가읽기및쓰기명 령을내릴수있는지결정하기위해스위칭패브릭내에서조정 (아비트레이션) 이필 요합니다.

12.2.3 IO 장치예제

목록 12.1는파이프라인상호연결의명세를구현하는 IO 장치를보여줍니다. 이 IO 장치는네개의로드가능한카운터를포함합니다. 이네개의카운터를주소로지 정하려면네개의주소비트가필요한데, 이는주소가바이트단위로계산되지만카운 터는 32 비트폭이기때문입니다. 읽기트랜잭션 (rd 가활성화됨) 으로카운터의 값을읽고, 다음클록사이클에그결과를 (dout 에서) 얻습니다. wr 을활성화하고 din 에값을설정하여카운터에씁니다.

지연된응답 (delayed acknowledge) 을구현하기위해, 단일비트레지스터

```
class CounterDevice extends Module {
  val io = IO(new PipeCon(4))

  val ackReg = RegInit(false.B)
  val addrReg = RegInit(0.U(2.W))
  val cntRegs = RegInit(VecInit(Seq.fill(4)(0.U(32.W))))

  ackReg := io.rd || io.wr
  when(io.rd) {
    addrReg := io.address(3, 2)
  }
  io.rdData := cntRegs(addrReg)

  for (i <- 0 until 4) {
    cntRegs(i) := cntRegs(i) + 1.U
  }
  when (io.wr) {
    cntRegs(io.address(3, 2)) := io.wrData
  }

  io.ack := ackReg
}
```

Listing 12.1: 네개의로드가능한카운터로구성된 IO 장치입니다.

주소	장치
0x0000–0x0fff	ROM
0x1000–0x1fff	RAM
0xf000	UART
0xf010	LED
0xf020	키

Table 12.1: 주소매핑예제입니다.

(ackReg) 를사용하여어서트된 rd 또는 wr 을지연시킵니다. 읽기결과는읽기명령다음클록사이클에제공되고주소는이명령사이클동안에만유효하므로, 주소를 addrReg 에저장해야합니다.

카운터자체는 4 개요소로이루어진작은레지스터파일 (Vec 의 Reg) 로구성됩니다. 카운터는 Scala Seq 를사용하여 0 으로초기화되는데, 이 Seq 는리셋값을 Chisel 상수로담아 fill 로생성됩니다. 그 Seq 가 VecInit 의입력이됩니다. 카운터는자유롭게동작하며새값이쓰이는경우를제외하고매클록사이클마다 1 씩증가합니다.

12.2.4 메모리매핑장치

저희예제시스템은모든메모리또는 IO 장치를공유주소선에연결합니다. 따라서이들은공유주소공간에나타납니다. 개별장치를선택하기위해, 상위비트중일부에대한주소디코딩을사용합니다. 이를메모리매핑장치라고부르며, 시스템설계의일부로서주소매핑을결정해야합니다.

표 12.1은 (16 비트) 마이크로컨트롤러에대한예제주소맵을보여줍니다. 16 비트주소를가정하므로, 주소의범위는 0x0000 에서 0xffff 사이입니다. 가장낮은주소 (실행할프로그램의시작지점) 에는프로그램을담고있는읽기전용메모리 (ROM) 를매핑합니다. 다음메모리영역에는데이터를위한쓰기가능한메모리 (RAM) 를매핑합니다. 저희는모든 IO 장치를주소공간의상위영역 (0xf000 이상) 에매핑하기로결정했는데, 이는나중에메모리를확장하고자할경우방해가되지않도록하기위해서입니다. 이예제에서는각 IO 장치에 16 바이트의주소를예약합니다. 이는임의로만든예제이며주소맵을결정할때모든자유도를가지고있다는점에유의하십시오.

일부 IO 장치는카운터장치처럼메모리매핑레지스터를가지지않고, 절 9.3에서설명한 ready/valid 인터페이스를가집니다. 예를들어절 11.2에서소개한 UART 는값을쓰기위한것과읽기위한것, 두개의 ready/valid 인터페이스를가

주소	I/O 장치	읽기	쓰기
0xf000	UART	상태	제어
0xf001	UART	수신버퍼	송신버퍼

Table 12.2: UART 에대한주소매핑입니다.

비트	상태	
0	TDRE	송신 (TX) 데이터레지스터비어있음
1	RDRF	수신 (RX) 데이터레지스터가득참

Table 12.3: 상태플래그입니다.

집니다. 일반적인해법은쓰기채널과읽기채널을하나의주소에매핑하고쓰기또는읽기명령에따라해당신호를구동하는것입니다. 저희는쓰기채널이새데이터워드를받을준비가되어있는지, 또는읽기채널에유효한데이터가있는지를알리기위해이두신호를다른주소의상태레지스터에매핑합니다.

표 12.2는 UART 에대한주소매핑을보여줍니다. 기본주소 (0xf000) 에서는읽기시상태레지스터에, 쓰기시에는선택적인제어레지스터에접근합니다. 다음주소 (0xf001) 에서는읽기버퍼로부터읽고송신버퍼에씁니다.

표 12.3는상태레지스터에매핑된두개의플래그를보여줍니다. 두비트모두쓰기또는읽기트랜잭션을수행할수있음을신호합니다. 전송데이터레지스터가비어있을때 (TDRE), 송신기 (TX) 에새데이터를쓸 (전송할) 수있습니다. 수신데이터레지스터가가득찼을때, 수신기 (RX) 에서데이터를읽을수있습니다. 이용어들은다소옛날용어를사용하는것처럼들릴수있습니다. 그리고이는우리의인터페이스에도해당합니다. 이는 8250 칩으로제작된 IBM PC 의첫번째직렬포트의매핑과정확히일치하며, 지금도여전히유효합니다.

폴링을위해상태레지스터에서 ready 와 valid 를사용하려면, 송신기의 ready 신호와수신기의 valid 신호는한번어서트된이후에는디어서트되어서는안된다는점에유의하십시오. 이것이보장될수없다면, 목록 9.7에나온것과같은두개의단일워드버퍼를 IO 인터페이스와 read/valid 인터페이스를가진장치사이에삽입할수있습니다.

TODO: 빈장치에서읽거나가득찬장치에쓸때어떤일이일어나는지에대한논의입니다.

저희의메모리매핑장치를위해, 번들을정의합니다:

```
class MemoryMappedIO extends Bundle {
  val address = Input(UInt(4.W))
  val rd = Input(Bool())
  val wr = Input(Bool())
  val rdData = Output(UInt(32.W))
  val wrData = Input(UInt(32.W))
  val ack = Output(Bool())
}
```

목록 12.2는직렬포트와같은스트리밍장치에대한메모리매핑인터페이스를 보여줍니다. 스트리밍장치는출력을위해 tx 에, 입력을위해 rx 에연결됩니다. mem 포트는파이프라인핸드셰이크를사용하여프로세서버스에연결됩니다.

이장치는파이프라인핸드셰이크에서정의된최소접근지연시간인한클록사이클의지연시간으로접근할수있습니다. 따라서저희는읽기또는쓰기가있었을때한 클록사이클뒤에 ack 신호를생성합니다 (ackReg).

상태레지스터 statusReg 는두개의플래그를담고있습니다: 하나는송신채널준비 (전송버퍼에공간이있음) 를위한것이고, 다른하나는수신채널유효 (수신버퍼에적어도한바이트가있음) 를위한것입니다.

읽기명령시, 저희는읽기값을반환할다음클록사이클에서사용하기위해읽기주소를레지스터 (addrReg) 에저장합니다. 주소에따라상태레지스터의값또는수신데이터중하나를반환합니다. 쓰기데이터는송신채널에직접연결되며, 쓰기신호 (wr) 는유효한입력임을알립니다.

코드예제를단순화하기위해, 수신채널에데이터가없는경우에도읽기로부터반환합니다. 대안으로는데이터가사용가능해질때까지 ack 를기다리게하는방법이 있을것입니다. 마찬가지로송신부분에서도, 저희는송신버퍼에공간이있는지여부를무시합니다. 송신버퍼에공간이생길때까지 ack 를어서트하지않음으로써쓰기를막을수도있을것입니다. 저희는이확인을값을읽거나쓰려하기전에상태레지스터를읽는소프트웨어에위임합니다.

TODO: Chisel 코드디코딩이지만, 아마도완전한시스템을구축할때 *Leros* 절에서다룰것입니다.

TODO: 다중마스터와범용인터커넥트 *(cloud)* 로의다음단계

12.3 버스및인터페이스표준

수년에걸쳐여러점대점방식및버스표준이제안되어왔습니다. 다음절에서는일반적인 SoC 상호연결표준을간략히개관합니다.

```
class MemMappedRV[T <: Data](gen: T, block: Boolean = false)
    extends Module {
  val io = IO(new Bundle() {
    val mem = new MemoryMappedIO()
    val tx = Decoupled(gen)
    val rx = Flipped(Decoupled(gen))
  })

  val statusReg = RegInit(0.U(2.W))
  val ackReg = RegInit(false.B)
  val addrReg = RegInit(0.U(1.W))
  val rdDlyReg = RegInit(false.B)

  statusReg := io.rx.valid ## io.tx.ready

  // ack
  ackReg := io.mem.rd || io.mem.wr
  io.mem.ack := ackReg

  // read from status or rx
  when (io.mem.rd) {
    addrReg := io.mem.address
  }
  rdDlyReg := io.mem.rd
  io.rx.ready := false.B
  when (addrReg === 1.U && rdDlyReg) {
    io.rx.ready := true.B
  }
  io.mem.rdData := Mux(addrReg === 0.U, statusReg,
      io.rx.bits)

  // write to tx
  io.tx.bits := io.mem.wrData
  io.tx.valid := io.mem.wr
}
```

Listing 12.2: ready/valid 장치를위한 IO 장치

12.3.1 Wishbone

Wishbone [16] 명세는고전적인의미의버스가아니라점대점통신을정의합니다. Wishbone 은여러오픈소스 IP 코어에서사용되는퍼블릭도메인표준입니다. Wishbone 인터페이스명세는여전히마이크로컴퓨터또는백플레인버스의전통을따르고있습니다. 그러나일반적으로점대점방식인[1] SoC 상호연결에는이것이최선의접근방식은아닙니다. 마스터는읽기또는쓰기사이클전체에걸쳐주소와데이터를유효하게유지하도록요구됩니다. 이는데이터가단한사이클동안만유효한마스터와의연결을복잡하게만듭니다. 이경우, 주소와데이터는 Wishbone 연결 이전에레지스터에저장되어야하거나, (시간과자원측면에서) 비용이많이드는멀티플렉서를사용해야합니다. 레지스터를사용하면한사이클의추가지연시간이발생합니다. 더나은접근방식은슬레이브에서주소와데이터를레지스터에저장하는것입니다. 이경우, 슬레이브에서의주소디코딩은주소가레지스터에저장되는사이클과동일한사이클에수행될수있습니다. 슬레이브의출력데이터에대해서도마스터와관련된유사한문제가존재합니다: 이데이터는단한사이클동안만유효하므로, 마스터가즉시읽지않는경우마스터가데이터를레지스터에저장해야합니다. 따라서슬레이브는 Wishbone 스트로브신호 (*wb.stb*) 가더이상할당되지않는경우에도마지막으로유효했던데이터를출력에유지해야합니다. 슬레이브에서데이터를유지하는것은일반적으로하드웨어복잡도측면에서 공짜입니다—이는 단지명세상의문제일뿐입니다. 고전적인 Wishbone 명세에는파이프라인읽기나쓰기를수행할방법이없습니다. 그러나최신 Wishbone 명세 (B4) 에는파이프라인정의도포함되어있습니다. 이제이명세에는반드시호환되지는않는두가지서로다른명세가포함되어있다는점에유의하십시오.

TODO: 주변장치용샘플코드, 어떻게테스트하는가? 그림설명 (다른책의텍스트)

12.3.2 AXI

Advanced Microcontroller Bus Architecture(AMBA) [2] 는 ARM 에서제공하는상호연결정의입니다. 이명세는세가지서로다른버스를정의합니다: Advanced High-performance Bus(AHB), Advanced System Bus(ASB), Advanced Peripheral Bus(APB) 입니다. AHB 는온칩메모리, 캐시, 외부메모리를프로세서에연결합니다. 주변장치는 APB 에연결됩니다. 브리지가 AHB 를더낮은대역폭의 APB 에연결합니다. AHB 버스전송은버스트동작을사용하면한사이클로이루어질수있습니다. APB 에서는버스전송에두사이클이필요하며, 버스트모드는사용할수없습니다. 대기상태가있는주변

[1] 여러슬레이브와마스터를연결하기위해버스대신멀티플렉서가사용됩니다.

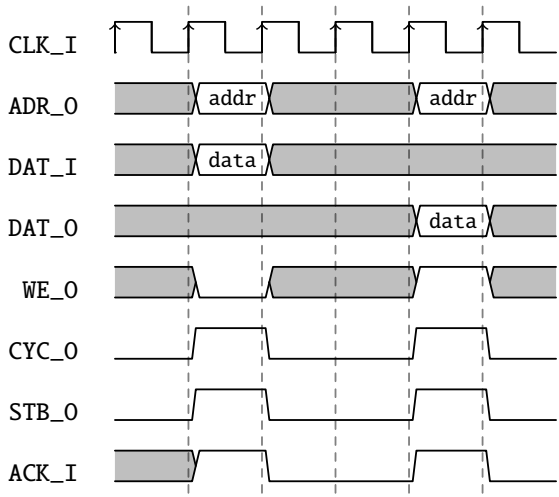


Figure 12.5: 비동기읽기에이어비동기쓰기를수행하는 Wishbone.

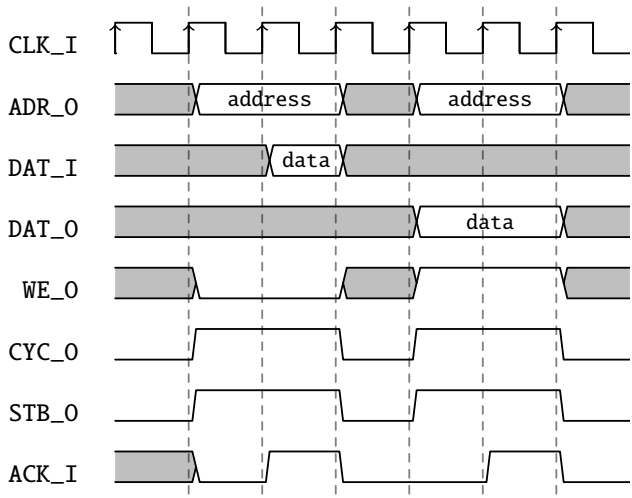


Figure 12.6: 동기읽기에이어동기쓰기를수행하는 Wishbone.

장치버스사이클은 APB 명세버전 3 에서추가되었습니다. ASB 는 AHB 의전신이며새로운설계에는권장되지않습니다 (ASB 는버스신호에클록의양쪽위상을모두사용하는데, 이는오늘날의동기식설계에서는매우이례적입니다).

Amba AXI(Advanced eXtensible Interface) 와 ACE 버전 4 [3] 는 AMBA 의최신확장판입니다. AXI 는 4 비트트랜잭션 ID 태그의도움으로순서가뒤바뀐 (out-of-order) 트랜잭션완료를도입합니다. ready 신호는트랜잭션의시작을확인해줍니다. 마스터는인터커넥트신호가준비될때까지트랜잭션정보 (예: 주소) 를유지해야합니다. 이러한개선은원래 AHB 명세의우아한단일사이클주소단계를망가뜨립니다.

AXI 버스는모든신호 (읽기주소, 읽기데이터, 쓰기주소, 쓰기데이터, 쓰기응답) 에대해 ready/valid 핸드셰이크를사용합니다. 쓰기주소와쓰기데이터의분리는도착하는주소와데이터의순서를임의로받아들일수있는더복잡한슬레이브를필요로합니다.

TODO: 서번트용샘플코드, 어떻게테스트하는가? 아마도 *Xilinx* 관련자료와 *zipcpu* 의코드를사용할수있을것입니다.

12.3.3 Open Core Protocol

Sonics Inc. 는 Open Core Protocol(OCP) [13] 를공개적으로자유롭게이용가능한개방형표준으로정의했습니다. 이표준은현재 OCP International Partnership(www.ocpip.org) 에서관리하고있습니다. Patmos 프로세서 [27] 와 T-CREST [21] 멀티코어플랫폼은 OCP 표준을사용합니다. Patmos 저장소[2]에는여러메모리컨트롤러, 다수의주변장치, 그리고 OCP 인터페이스를갖춘네트워크온칩이포함되어있습니다.

TODO: 예시로 *Patmos* 에서코드일부를가져올것입니다.

12.3.4 그밖의버스명세

Avalon [1] 인터페이스사양은 system-on-a-programmable-chip 인터커넥션을위해 Intel 이제공합니다. Avalon 은직접정적 RAM 연결을위한단순한비동기식인터페이스부터가변지연시간을갖는정교한파이프라인전송에이르기까지광범위한인터커넥션장치를정의합니다. 이러한유연성은주변장치를 Avalon 에연결하는쉬운경로를제공합니다. 이러한유연성이어떻게가능할까요? *Avalon* 스위치패브릭이이모든서로다른인터커넥션유형사이를변환합니다. 스위치패브릭은 Intel 의 SOPC Builder 도구에의해생성됩니다. 그러

[2] https://github.com/t-crest/patmos

나이스위치패브릭은 Intel 의독점적인것으로보이며, 따라서이사양을 Intel FPGA 에종속시킵니다.

On-Chip Peripheral Bus(OPB) [11] 는 IBM 이제공한개방형표준으로, 몇년전 Xilinx 에서사용된바있습니다. OPB 는여러마스터와슬레이브를위한버스를규정합니다. 이버스의구현방식은명세에서직접정의되어있지않습니다. 분산링, 중앙집중식멀티플렉서, 또는중앙집중식 AND/OR 네트워크가제안됩니다. Xilinx 는 AND/OR 방식을사용했으며, 모든마스터와슬레이브는비활성상태일때데이터버스를 0 으로구동해야했습니다. Xilinx 는이후모든상호연결을 AXI 로전환했습니다.

12.4 연습문제

Listing 12.2에나온코드와 rx 및 tx 로향하는스트리밍장치를사용하십시오. 또는그러한장치에대한 ChiselTest 시뮬레이션을작성하십시오. 메모리인터페이스를테스트하는테스트벤치를작성하십시오. 테스트가상태플래그를무시할경우, 즉유효하지않은데이터를읽거나송신채널이준비되지않았을때쓰기데이터를누락시킬경우어떤일이일어나는지탐구하십시오. ack 신호가스트리밍장치가 ready 또는 valid 상태가될때까지지연되도록 MemeoryMappedRV 모듈을갱신하십시오. 여러분의테스트벤치는지연된 ack 를처리할수있습니까? 스트리밍장치와메모리인터페이스를함께시뮬레이션할경우 Scala 코드가다소어색해지지않습니까—두장치를병렬로처리하기위해두개의소프트웨어상태기계가필요하지않습니까? 이문제는테스트장에서설명한것처럼멀티스레드테스트를사용하면우아하게해결할수있습니다.

13 디버깅, 테스트, 검증

제 3장에서는 Chisel 설계를테스트하는방법에대해간략히소개했습니다. 이장에서는테스트와검증을더깊이다루겠습니다.

테스트와검증은소프트웨어개발에서와디지털설계에서다소다른의미를갖습니다. 소프트웨어개발에서테스트는컴포넌트에대해테스트를실행하는것을의미하는반면, 검증은보통정형검증 (수학적증명이나모델체킹을이용한전수테스트)의줄임말입니다. 디지털설계에서는 테스트벤치를작성하여 피시험장치 (DUT)를자극하고검사할때소프트웨어와유사한의미로테스트라는용어를사용합니다. 그러나테스트라는용어는 (내장자체테스트를이용하는테스터상에서) 물리적칩을실제로테스트하는것을가리킬때도사용됩니다. 따라서디지털설계커뮤니티는하드웨어기술을테스트하는것을가리킬때점차검증이라는용어를사용하는방향으로나아가고있습니다. 하드웨어컴포넌트를검증하기위해정형방법을적용하고자할경우, 이를정형검증이라고부릅니다. 이책에서는하드웨어기술에대한테스트를작성할때 테스트라는용어를계속사용하겠습니다. 검증 (테스트) 은시뮬레이터상에서설계를실행하는동적검증으로수행하거나, 모델체커나 SMT 솔버를이용한정형검증으로수행할수있습니다.

13.1 디버깅

설계및코딩단계에서는자주설계를디버깅하게됩니다. 디버깅은코드에서결함을찾아내는과정입니다. 이러한결함은 버그라고불립니다. 디버깅은흔히새로운코드를작성하는것과병행하여수행됩니다.

프로그램을디버깅하는방법에는디버거를사용하거나, 단순히관심있는값을터미널에출력하는방법이있는데, 이를 printf 디버깅이라고부릅니다. 하드웨어에서는요소들이병렬로 실행됩니다. 따라서하드웨어디버깅의일반적인형태는파형을생성하여관심있는신호가시간에따라어떻게변화하는지관찰하는것입니다. 이를 파형디버깅이라고부릅니다.[1]

Chisel 테스터는파형을생성할수있으며, 이는예를들어 GTKWave로볼수있습니다. 하지만빠른확인을위해회로시뮬레이션중에신호값을출력하는것도가능

[1] 이에대한위키백과항목은없는데, 하나만들어야합니다.

합니다. 값은클록의상승에지에서출력됩니다.

13.2 Chisel 에서의테스트

ChiselTest 는 ScalaTest 를기반으로합니다. 따라서간단한 sbt test 명령으로모든테스트를실행할수있습니다. ScalaTest 는기본적으로멀티스레드테스트도지원하므로, 프로젝트에여러테스트클래스가있다면이를병렬로실행할수있습니다. 또한 FlatSpec 문법을사용하여명확한테스트설명을작성하고디버깅을더쉽게만들수있습니다.

Chisel 테스트는 ChiselScalatestTester 트레이트와함께 AnyFlatSpec 를확장하는클래스로정의합니다. ChiselTest 는 DUT 의 IO 포트를대상으로동작하는 peek, poke, expect, step 메서드를사용합니다. ChiselTest 메서드는 Chisel 타입 (즉, UInt, SInt, Bool) 을대상으로동작합니다. 하지만값을 peek 할때는 Scala 로작성된테스트를위해보통 Scala 타입을원하게됩니다. 따라서두가지추가메서드가존재하는데, (1) peekInt() 는 Scala Int 를반환하고, (2) peekBoolean() 은 Scala Boolean 을반환합니다. 시뮬레이션을한클록사이클진행시키려면 DUT 의암묵적클록포트에대해 step() 을호출합니다.

test() 함수안에서테스트를정의할수있으며, 이함수는테스트할모듈을매개변수로받습니다. 예를들어, 다음테스터는 BCD 테이블에대한몇가지입력을테스트합니다.

```
class BcdTableTest extends AnyFlatSpec with
    ChiselScalatestTester {
  "BCD table" should "output BCD encoded numbers" in {
    test(new BcdTable) { dut =>
      dut.io.address.poke(0.U)
      dut.io.data.expect("h00".U)
      dut.io.address.poke(1.U)
      dut.io.data.expect("h01".U)
      dut.io.address.poke(13.U)
      dut.io.data.expect("h13".U)
      dut.io.address.poke(99.U)
      dut.io.data.expect("h99".U)
    }
  }
}
```

대안으로, behavior of “module name” 문법을사용하여 it 으로모듈을참조할수있습니다. 이는하나의모듈에대해여러테스트가있을때유용합니다.

```
class BcdTableTest extends FlatSpec with
    ChiselScalatestTester {
  behavior of "BCD table"

  it should "output BCD encoded numbers" in {
    test(new BcdTable) { dut =>
      ...
    }
  }
}
```

간단한테스트는 poke 로 DUT 에테스트벡터를작성하고, 클록을진행시킨후, expect 로출력을검사하는방식으로시작합니다. 디버깅목적으로는값을 peek 하여수동으로확인할수있도록출력할수도있습니다. Listing 13.1의코드는예제 IO 장치로 Chapter 12에서소개한카운터장치를테스트합니다.

13.2.1 함수사용하기

보시다시피이테스트는몇가지경우만다루는데도이미읽기에매우깁니다. 이모든 poke 와 expect 들은번거롭습니다. 첫단계로, 읽기요청과쓰기요청을나타내는함수를도입하겠습니다. 이함수들은테스트코드에서인터페이스핀을수동으로 “비트뱅잉”하는작업을추상화합니다. Listing 13.2은이러한함수를사용한테스트를보여줍니다. 간편함을위해클록을진행시키는 step 함수도정의합니다.

read 함수는주소를매개변수로받아읽은값을반환합니다. 주소와읽기신호를 poke 한후, 이함수는클록을한사이클진행시키고읽기신호를해제합니다. 우리예제장치에서는읽은값이한클록사이클후에사용가능해야합니다. 하지만지연시간이더긴장치를위해서도 read 함수를일반화하여, read 함수가 ack 가참이될때까지무한루프에서대기하도록합니다. Scala Boolean 을읽기위해 peekBoolean 을사용한다는점에유의하십시오. 하지만장치가요청후에 ack 를결코어서트하지않는오류가있다면, 테스트는무한루프에갇히게됩니다. 더견고한 read 함수는 ack 폴링에타임아웃을포함해야합니다. 마지막으로, rdData 에서데이터를 peekInt() 로읽어 Scala 정수값 (구체적으로는임의크기의정수를표현하기위한 BigInt) 을얻습니다.

write 함수는주소와데이터매개변수를 Scala Int 로받습니다. 읽기함수와마찬가지로, 값들이장치에 poke 되고, 클록이한클록사이클만큼진행되며, 쓰기신호가비활성화됩니다. 또한 ack 가참이될때까지무한루프에서대기합니다.

```
  "CounterDevice" should "work" in {
    test(new CounterDevice()) { dut =>
      dut.io.ack.expect(false.B)
      dut.clock.step()
      dut.io.address.poke(0.U)
      dut.io.rd.poke(true.B)
      dut.io.ack.expect(false.B)
      dut.clock.step()
      dut.io.rd.poke(false.B)
      dut.io.ack.expect(true.B)
      dut.clock.step(100)
      dut.io.rd.poke(true.B)
      dut.io.address.poke(4.U)
      dut.clock.step()
      assert(dut.io.rdData.peekInt() > 100)
      dut.io.wr.poke(true.B)
      dut.io.wrData.poke(0.U)
      dut.clock.step()
      dut.io.wr.poke(false.B)
      dut.io.rd.poke(true.B)
      dut.clock.step()
      dut.io.rdData.expect(1.U)
      dut.io.address.poke(0.U)
      dut.clock.step()
      assert(dut.io.rdData.peekInt() > 100)
    }
  }
```

Listing 13.1: 카운터장치테스트하기.

```
"CounterDevice" should "work with functions" in {
  test(new CounterDevice()) { dut =>

    def step(n: Int = 1) = {
      dut.clock.step(n)
    }
    def read(addr: Int) = {
      dut.io.address.poke(addr.U)
      dut.io.rd.poke(true.B)
      step()
      dut.io.rd.poke(false.B)
      while (!dut.io.ack.peekBoolean()) {
        step()
      }
      dut.io.rdData.peekInt()
    }
    def write(addr: Int, data: Int) = {
      dut.io.address.poke(addr.U)
      dut.io.wrData.poke(data.U)
      dut.io.wr.poke(true.B)
      step()
      dut.io.wr.poke(false.B)
      while (!dut.io.ack.peekBoolean()) {
        step()
      }
    }

    for (i <- 0 until 4) {
      assert(read(i*4) < 10, s"Counter $i should have just
          started")
    }
    step(100)
    for (i <- 0 until 4) {
      assert(read(i*4) > 100, s"Counter $i should advance")
    }
    write(2*4, 0)
    write(3*4, 1000)
    assert(read(2*4) < 5, "Counter should reset")
    assert(read(3*4) > 1000, "Counter should load")
  }
}
```

Listing 13.2: 함수를사용해카운터장치테스트하기.

이세가지함수를사용할수있게되면, 더적은줄의코드로더읽기쉬운테스트를작성할수있습니다. 이테스트코드는이미원래의비트뱅잉테스터보다더많은경우를다루고있습니다.[2]

13.2.2 테스트선택하기

테스트스위트가크다면, 지속적통합 실행의일부로테스트의일부집합만실행하고싶을수있습니다. 이를달성하면서도단일 SBT 명령만실행하면되게하는가장쉬운방법은테스트에태그를붙이는것입니다.

```
object Unnecessary extends Tag("Unnecessary")

class TagTest extends AnyFlatSpec with Matchers {
  "Integers" should "add" taggedAs(Unnecessary) in {
    17 + 25 should be (42)
  }
}
```

기본적으로모든테스트는 sbt test 또는 sbt testOnly * 로실행됩니다. 예를들어 Unnecessary 로태그된테스트를제외하려면다음을실행하면됩니다.

```
$ sbt "testOnly * -- -l Unnecessary"
```

이명령을실행하면, 해당테스트는터미널에무시됨 (ignored) 으로표시됩니다.

```
[info] TagTest:
[info] Integers
...
[info] No tests were executed.
```

테스트 (및태그) 가패키지의일부라면, 반드시둘다에대한전체참조경로를제공해야한다는점을기억하십시오.

13.2.3 내부신호접근하기

회로를테스트할때, 테스트코드는일반적으로 DUT 의포트에만접근할수있습니다. 이러한추상화는대체로좋은관행인데, 내부신호와상태에접근하는것은 (하드웨어와소프트웨어테스트모두에서) 나쁜관행으로간주되기때문입니다.

[2] 실제로저는카운터장치에서오프바이원오류 (until 4 대신 until 3) 를범한적이있는데, 이는더포괄적인두번째테스트에서야발견했습니다.

```
class TickGen extends Module {
  val io = IO(new Bundle {
    val tick = Output(Bool())
  })

  val cntReg = RegInit(0.U(8.W))
  cntReg := cntReg + 1.U
  io.tick := cntReg === 9.U
  when(io.tick) {
    cntReg := 0.U
  }
}
```

Listing 13.3: DUT 로서의틱생성기.

그러나때로는내부상태에접근하는것이유익할수있습니다. 예를들어, 작은어셈블러테스트프로그램으로마이크로프로세서를테스트하는경우입니다. 이경우, 일반적으로프로세서의하드웨어구현상태를동일프로세서의소프트웨어기반시뮬레이터와비교하게됩니다. RISC 스타일프로세서의경우, 계산되거나로드되거나저장되는모든데이터가언젠가는레지스터파일을거치므로, 두구현의레지스터파일내용을비교하는것만으로도이런종류의테스트에는충분합니다. 또다른사용사례는내부상태에접근하여 (데이터패스가있거나없는) 상태기계를탐색하고테스트하는것입니다.

TODO: Leros 챕터에서코시뮬레이션을설명하고탐구합니다.

내부신호에접근하는것을실제로보여주기위해, 우리는매우최소한의예제를사용합니다: 내부카운터를가진틱생성기입니다. 리스팅 13.3이그코드를보여줍니다. 필요한신호 tick 만출력포트에연결되어있습니다. 카운터는노출되지않는데, 이는좋은설계관행입니다. 예를들어, 우리는테스트코드에서이내부카운터에접근하고자합니다. 카운터를노출하기위해포트를추가할수도있습니다. 심지어 Scala Boolean 플래그를사용하여, 디버깅할때는이포트를조건부로두고하드웨어를생성할때는비활성화할수도있습니다. 그러나디버깅코드를하드웨어기술에섞는것은좋은관행이아닙니다.

내부신호에접근하기위해, 우리는 BoringUtils 를사용할수있습니다. BoringUtils 는모듈계층구조를관통하여연결을 뚫는것을가능하게합니다. 배후에서는계층구조전체에걸쳐필요한추가포트들을덧붙입니다. 이는우리가수동으로할수있는것과정확히동일한작업을수행하지만, 원본코드가지저분해지는것을피할수있습니다.

```
class TickGenTestTop extends Module {
  val io = IO(new Bundle {
    val tick = Output(Bool())
    val counter = Output(UInt(8.W))
  })

  val tickGen = Module(new TickGen)
  io.tick := tickGen.io.tick
  io.counter := DontCare
  BoringUtils.bore(tickGen.cntReg, Seq(io.counter))
}
```

Listing 13.4: 우리 DUT 를위한최상위래퍼.

이글을쓰는시점에서, BoringUtils 는여전히실험적인것으로간주됩니다. 따라서이를사용할때는다음패키지를포함해야합니다:

```
import chisel3.util.experimental.BoringUtils
```

추가포트를지원하기위해, 우리는테스트를위해 DUT 를또다른최상위모듈로감쌉니다. 리스팅 13.4은추가포트인카운터를가진그최상위모듈을보여줍니다. TickGenTestTop 내에서, 우리는원본 DUT 를인스턴스화하고 tick 포트를연결합니다. counter 출력의경우, 먼저 Chisel 컴파일러를만족시키기위해값을할당해야합니다. 나중에내부모듈에연결할것이므로, 일단 DontCare 에연결합니다. 다음줄은 io.counter 를 tickGen 의카운트레지스터에연결합니다. boring() 이여러신호에대한연결을지원하므로, io.counter 를 Scala Seq 로감싸야합니다.

마지막으로, 리스팅 13.5는우리 DUT 를위한테스트코드를보여줍니다. 추가된출력포트를사용하기위해최상위래퍼를인스턴스화한다는점에유의하십시오.

13.2.4 멀티스레드테스트

디지털하드웨어는본질적으로병렬적입니다. 이러한병렬회로를테스트하기위해테스트코드에서병렬성을표현하는것역시도움이됩니다. 예를들어, 한스레드는회로에데이터를채워넣고다른스레드는그회로의출력을확인합니다. 단일스레드에서도이를수행할수있지만, 그럴경우클록의진행을공유하는하나의함수에서로다른작업을위한코드를섞어야합니다. 멀티스레드테스트를사용하면, 각스레드

```
  test(new TickGenTestTop()) { dut =>
    dut.io.tick.expect(false.B)
    dut.io.counter.expect(0.U)

    dut.clock.step()
    dut.io.tick.expect(false.B)
    dut.io.counter.expect(1.U)

    dut.clock.step(8)
    dut.io.tick.expect(true.B)
    dut.io.counter.expect(9.U)

    dut.clock.step()
    dut.io.tick.expect(false.B)
    dut.io.counter.expect(0.U)
  }
```

Listing 13.5: 내부신호에접근하여 DUT 테스트하기.

가독립적으로클록을진행시킬수있습니다. 스레드들은 clock() 호출시내부적으로동기화됩니다.

ChiselTest 는 fork 와 join 호출을사용하여멀티스레드테스트를지원합니다. fork 는테스트코드블록을매개변수로받아새로운테스터스레드를생성하며, join 은 (fork 호출이반환한) 테스터스레드변수에대해호출되어해당스레드가메인스레드에합류할때까지기다릴수있습니다.

여러스레드를실행하는것은 peek 과 poke 에몇가지새로운제약을가하는데, 두스레드가동시에동일한신호를 peek(각각 poke) 할수없다는것입니다. 마찬가지로, 올바른동작을보장하기위해스레드들은 step 호출시동기화됩니다. 다음코드는한스레드에서요소를큐에넣고메인스레드에서그것을큐에서빼내는 FIFO 의작은테스트입니다:

```
it should "work with multiple threads" in {
  test(new BubbleFifo(8, 4)) { dut =>
    val enq = fork {
      while (dut.io.enq.full.peekBoolean())
        dut.clock.step()
      dut.io.enq.din.poke(42.U)
      dut.io.enq.write.poke(true.B)
```

```
        dut.clock.step()
        dut.io.enq.write.poke(false.B)
      }
      while (dut.io.deq.empty.peekBoolean())
        dut.clock.step()
      dut.io.deq.dout.expect(42.U)
      dut.io.deq.read.poke(true.B)
      dut.clock.step()
      dut.io.deq.empty.expect(true.B)
      enq.join()
    }
  }
```

여러스레드는중첩된 fork 호출을통해생성됩니다. 생성된스레드들은첫번째스레드가이후의어떤스레드보다도먼저끝나서는안되는계층구조를나타냅니다.

13.2.5 시뮬레이터백엔드

기본적으로, ChiselTest 로작성된테스트는 Treadle 시뮬레이션백엔드로실행됩니다. Treadle 의장점은빠른시작시간과, 추가도구를설치할필요가없다는점입니다.

그러나대규모시스템테스트는예를들어래치를지원하기위해다른백엔드를필요로하거나, 시뮬레이션시간측면에서이득을볼수도있습니다. 이를가능하게하기위해, ChiselTest 는두가지다른백엔드를지원합니다: Verilator와 Synopsys VCS. Verilator 는오픈소스이므로, 이절에서제시하는예제에는이것을사용하겠습니다. 제시된모든경우에서 VCS 가 Verilator 의대안이될수있다는점에유의하십시오.

파형절에서보았듯이, 다른백엔드로전환하려면 withAnnotations 호출에다른어노테이션을추가하면됩니다. Verilator 를사용하려면, 다음어노테이션을추가하십시오:

```
test(new
    Dut()).withAnnotations(Seq(VerilatorBackendAnnotation))
    {
```

추가적인유연성은백엔드를시작하는시뮬레이터명령에여러분의스위치를제공함으로써얻어집니다. 이는 Verilator 시뮬레이션명령에스위치를추가하려면 VerilatorFlags 를, GCC 에스위치를추가하려면 VerilatorCFlags 를사용함으로써이루어집니다. 이들은백엔드어노테이션과함께어노테이션목록에포함되어

야합니다. 명령줄인자의상세한목록을찾으려면해당도구의사용자매뉴얼을참조해야합니다. VerilatorFlags 와 VerilatorCFlags 어노테이션은고급기능이며일반적으로는필요하지않다는점에유의하십시오. 게다가, 이플래그들이계속안정적으로유지된다는보장은없습니다.

ChiselTest 0.3.4 이상에서는시뮬레이션에서코드커버리지측정을직접지원한다는점에유의하십시오. 이를지원하려면 Verilator 4.028 버전이상을설치하십시오.

또한시뮬레이터마다동작방식이다르다는점에유의하십시오. Verilator 는동기식시뮬레이터로, 클록의상승에지에서만업데이트를실행하므로래치를지원하지않습니다. 또한공식적으로는다중클록을지원하지않습니다. 반면 VCS 는이벤트기반시뮬레이터로, 시뮬레이션이훨씬더상세하며합성가능한모든 Verilog 구성요소를지원합니다. 일반적으로단일클록회로에서는 Verilator 가가장빠르고가장널리사용가능한도구입니다.

13.3 어서션과정형검증

프로그래밍언어에서어서션문은프로그램에대한가정을명시할수있게해줍니다. 어서션조건이 false 로평가되면, 프로그램은보통예외와함께중단됩니다. Chisel 도하드웨어에서가정을명시하기위한어서션을지원합니다. 이러한어서션은디지털설계를시뮬레이션할때검사됩니다. 조건이 false 로평가되면시뮬레이션은오류메시지와함께중단됩니다. 어서션은하드웨어를생성할때는무시되는데, 하드웨어가실패한어서션을전달할 (쉬운) 방법이없기때문입니다.

리스트 13.6는어서션사용예시를보여줍니다. 이는사용법을보여주기위한사소한어서션입니다. 만약실수로덧셈대신뺄셈을사용하거나나중에 sum 을다른값으로재할당하는오류를범한다면, 테스트중에그오류를잡아내어아래와유사한오류메시지를얻게될것입니다:

```
Assertion failed
    at Assert.scala:20 assert(sum <= a + b)
```

하지만주석에서읽을수있듯이, 처음두어서션은항상참은아니므로사용할수없습니다. 우리는이를정형검증을통해알아냈습니다. 저는모든어서션을모듈의끝에배치할것을제안하는데, 그래야의도된설계를읽는데방해가되지않기때문입니다.

TODO: 경계조건을간과하는것에대해쓰십시오. 그저너무작은숫자만사용하고오버플로는테스트하지마십시오.

어서션은시뮬레이션중에실행되며, 우리는이를위한테스트케이스를작성해야합니다. 하지만모든가능성을유발하는테스트를작성하는것은어렵습니다 (불가

```
class Assert extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(8.W))
    val b = Input(UInt(8.W))
    val sum = Output(UInt(8.W))
  })
  io.sum := io.a + io.b

  /* the following will not be true when
  the addition overflows
  assert(io.sum >= io.a)
  assert(io.sum >= io.b)
   */
  assert(io.sum === io.a + io.b)
}
```

Listing 13.6: Chisel 에서어서션사용하기.

능합니다). 간과된경계사례의오류를잡아내기위해정형검증을사용할수있습니다. Kevin Laeufer 는 ChiselTest 에정형검증을추가했습니다 [12]. 정형검증에는바로그동일한어서션이사용됩니다.

Chisel 로정형검증을탐구하려면, 오픈소스 Z3 정리증명기를설치하십시오. 정형검증을사용하려면 test(..) 를 verify(..) 로대체합니다. 리스트 13.7는 (순진한) 어서션을사용하여간단한가산기회로에대해정형검증을실행하는방법을보여줍니다.

Chisel formal 이회로또는어서션의오류를즉시찾아냈다는점이저자를놀라게했습니다. 문제를조사하기위해, 어서션위반을유발하는입력데이터를찾기위해파형을살펴볼수있습니다. 입력으로 0xdb 와 0x65 를사용하면합이 0x40 이됩니다. 이입력값은 8 비트덧셈의오버플로를유발했습니다. 우리의간단한테스트케이스에서는오버플로값을테스트하는것을놓쳤습니다. 이검증은합이입력보다크거나같다는어서션이가능한오버플로로인해잘못되었음을보여주었습니다.

TODO: https://github.com/tdb-alcorn/chisel-formal에서계속읽으십시오. 이것이 *Kevin* 의작업과어떤관계가있습니까*?*

TODO: 한계는무엇입니까*?*

```
  "AssertTest" should "pass" in {
    verify(new Assert(), Seq(BoundedCheck(5),
        WriteVcdAnnotation))
  }
```

Listing 13.7: 회로를정형검증하기.

13.4 연습문제

익스트림프로그래밍은빠른반복주기와단위테스트에대한강한의존성에초점을맞춘애자일소프트웨어개발스타일입니다. 순수한형태에서는기능을구현하기전에 먼저테스트를작성합니다. 이스타일은실제로는그리자주사용되지않습니다. 하지만이를탐구해보는것은테스트를산출물개발의중요한부분으로집중하는데도움이될수있습니다.

따라서제안하는연습문제는아직구현하지않은설계에대한테스트벤치를작성하는것입니다. 제 7장에서작은프로젝트하나, 예를들어디바운싱회로나다수결기반필터링설계를골라그에대한테스트를작성하십시오. 그런다음, 하드웨어설계자체를구현하십시오.

이작은실험의경험을탐구해보십시오. 여러분의설계에서오류를찾아낸테스트를구현했습니까? 모든테스트가통과한다면, 합리적인설계공간을포괄하는테스트를가지고있다고확신할수있습니까? 여러분의테스트를어떻게테스트합니까? DUT 에결함을추가하고테스트가이를잡아내는지확인해보십시오.

이연습문제를진행하면서, 테스트가어렵고모든오류를잡아내는것은아마도불가능하다는불쾌한느낌을경험할수있습니다.[3] 하지만테스트를보완하기위한정형검증의최근발전에희망이있습니다. Chisel 을이용한정형검증이라는주제는이책의향후개정판에서확장될예정입니다.

[3] Dijkstra 의유명한인용문은다음과같습니다. " 프로그램테스트는버그의존재를보여줄수는있지만, 결코버그의부재를보여줄수는없다!"

14 프로세서의설계

이책의마지막장중하나로서, 우리는중간규모의프로젝트를제시합니다: 마이크로프로세서의설계, 시뮬레이션, 테스트입니다. 이프로젝트를관리가능한수준으로유지하기위해, 우리는간단한누산기머신을설계합니다. 이프로세서는 Leros [25] 라고불리며, https://github.com/leros-dev/leros에서오픈소스로이용할수있습니다. 이것은고급예제이며, 제시된코드예제를따라가려면어느정도의컴퓨터구조지식이필요하다는점을언급하고자합니다.

14.1 명령어집합구조

명령어집합의정의는명령어집합구조 (ISA) 라고도불립니다. ISA 는컴퓨터구조에서가장중요한추상화입니다. ISA 는컴파일러 (또는어셈블러프로그래머) 와구체적인프로세서구현사이의계약입니다. ISA 는실제구현과는독립적입니다. 마이크로프로세서 (마이크로아키텍처라고도불림) 의서로다른구현들이동일한 ISA 를실행할수있습니다.

Leros 는단순하게설계되었지만, 여전히 C 컴파일러의좋은대상입니다. 명령어에대한설명은한페이지에담깁니다. 표 14.1를참조하십시오.

Leros 는누산기기계입니다. 이는모든연산이누산기를소스입력중하나로가지며, 결과는보통누산기에기록됨을의미합니다. 산술또는논리연산의두번째피연산자는즉시값 (상수) 이거나 256 개의온칩레지스터중하나에서가져올수있습니다. 메모리접근 (로드또는스토어) 역시누산기를통해수행됩니다. 로드시에는메모리에서가져온값이누산기에저장되며, 스토어시에는값을누산기에서가져옵니다. 메모리접근을위한주소는주소레지스터 (AR) 에저장됩니다.

프로그램카운터 (PC) 는명령어메모리내의현재명령어를가리킵니다. PC 는보통다음명령어로증가합니다. 제어흐름을구현하기위해, PC 는분기또는점프명령어에의해조작될수있습니다. Leros 에는무조건분기명령어와조건분기명령어가있습니다. 조건분기는누산기의내용에따라결정됩니다. 예를들어, brz 는누산기의내용이 0 일때만분기합니다. Leros 의분기는현재명령어를기준으로상대적이며, 약 2000 개의명령어만큼앞뒤로분기할수있습니다. 더큰규모의제어흐름변경과함수호출및반환을위해, Leros 에는 jump-and-link(jal) 명령어가있습니다. 이명령어는누산기에있는주소로점프하며, 다음명령어의주소를레

연산코드	기능	설명
add	A = A + Rn	레지스터 Rn 을 A 에더합니다
addi	A = A + i	즉치값 i 를 A 에더합니다
sub	A = A - Rn	A 에서레지스터 Rn 을뺍니다
subi	A = A - i	A 에서즉치값 i 를뺍니다
shr	A = A >>> 1	A 를논리적으로오른쪽으로시프트합니다
load	A = Rn	레지스터 Rn 을 A 에로드합니다
loadi	A = i	즉치값 i 를 A 에로드합니다
and	A = A and Rn	레지스터 Rn 을 A 와 And 연산합니다
andi	A = A and i	즉시값 i 를 A 와 And 연산합니다
or	A = A or Rn	레지스터 Rn 을 A 와 Or 연산합니다
ori	A = A or i	즉시값 i 를 A 와 Or 연산합니다
xor	A = A xor Rn	레지스터 Rn 을 A 와 Xor 연산합니다
xori	A = A xor i	즉시값 i 를 A 와 Xor 연산합니다
loadhi	A_{15-8} = i	두번째바이트에즉시값을적재합니다
loadh2i	A_{23-16} = i	세번째바이트에즉시값을적재합니다
loadh3i	A_{31-24} = i	네번째바이트에즉시값로드
store	Rn = A	레지스터 Rn 에 A 저장
jal	PC = A, Rn = PC + 2	A 로점프하고반환주소저장
ldaddr	AR = A	주소레지스터 AR 에 A 로드
loadind	A = mem[AR+(i << 2)]	메모리에서워드를 A 로로드
loadindb	A = mem[AR+i]_{7-0}	메모리에서바이트를 A 로로드
loadindh	A = $\text{mem[AR+(i << 1)]}_{15-0}$	메모리에서하프워드를 A 로로드
storeind	mem[AR+(i << 2)] = A	A 를메모리에저장
storeindb	$\text{mem[AR+i]} = A_{7-0}$	메모리에바이트를저장합니다
storeindh	$\text{mem[AR+(i << 1)]} = A_{15-0}$	메모리에하프워드를저장합니다
br	PC = PC + o	분기
brz	if A == 0 PC = PC + o	A 가 0 이면분기합니다
brnz	if A != 0 PC = PC + o	A 가 0 이아니면분기합니다
brp	if A >= 0 PC = PC + o	A 가양수이면분기합니다
brn	if A < 0 PC = PC + o	A 가음수이면분기합니다
scall		시스템호출 (시뮬레이션훅)

Table 14.1: Leros 명령어집합.

지스터에저장합니다. 이값은이후 jal 을이용해함수에서복귀하는데사용될수있습니다.

누산기와레지스터파일은현재구현에서 32 비트폭입니다.[1]

표 14.1는 Leros 의명령어집합을보여줍니다. A 는누산기를나타내고, PC 는프로그램카운터이며, i 는즉시값 (0 에서 255 까지) 이고, Rn 은레지스터 n(0 에서 255 까지) 이며, o 는 PC 를기준으로한분기오프셋이고, AR 은메모리접근을위한주소레지스터입니다.

다음코드스니펫은어셈블리로작성된 Leros 명령어의예시를보여줍니다.

```
loadi 1
addi 2
ori 0x50
andi 0x1f
subi 0x13
loadi 0xab
addi 0x01
subi 0xac

scall 0
```

각명령어는명령어이름 (오피코드니모닉이라고도함) 과상수로구성됨을알수있습니다. 상수는 10 진법또는 16 진법표기로작성될수있습니다. 이코드는로드, 산술, 논리명령어의즉시값버전을보여줍니다. 마지막명령어 (scall 0) 는시스템호출이며실행 (또는시뮬레이션) 을종료합니다. 이짧은프로그램은 Leros 테스트스위트의일부입니다. 테스트의관례상, 프로그램이끝날때누산기는 0 을포함해야합니다.

명령어는 16 비트폭입니다. 상위바이트는명령어를인코딩하는데사용되며, 하위바이트는즉시값, 레지스터번호, 또는분기오프셋중하나를포함합니다 (분기오프셋의일부는상위바이트의비트도사용합니다). 예를들어, 00001001.00000010 은누산기에 2 를더하는 add immediate 명령어인반면, 00001000.00000011 은 R3 의내용을누산기에더합니다. 분기의경우, 더큰오프셋을위해명령어비트중 3 개를사용합니다.

목록 14.1은각명령어의상위 8 비트에서의인코딩을보여줍니다. 현재모든명령어비트가사용되지는않습니다 (사용되지않는비트는 -로표시됩니다).

[1] 16 비트또는 64 비트버전의 Leros 도구현할수있도록설정가능하게유지하려합니다.

```
+--------+----------+ |00000---| nop | |000010-0| add |
    |000010-1| addi | |000011-0| sub | |000011-1| subi |
    |00010---| sra | |00011---| - | |00100000| load |
    |00100001| loadi | |00100010| and | |00100011| andi |
    |00100100| or | |00100101| ori | |00100110| xor |
    |00100111| xori | |00101001| loadhi | |00101010| loadh2i
    | |00101011| loadh3i | |00110---| store | |001110-?| out
    | |000001-?| in | |01000---| jal | |01001---| - |
    |01010---| ldaddr | |01100-00| ldind | |01100-01| ldindb
    | |01100-10| ldindh | |01110-00| stind | |01110-01|
    stindb | |01110-10| stindh | |1000nnnn| br | |1001nnnn|
    brz | |1010nnnn| brnz | |1011nnnn| brp | |1100nnnn| brn
    | |11111111| scall | +--------+----------+
```

Listing 14.1: Leros instruction encoding.

14.2 데이터패스

9.2절에서는상태기계와데이터패스를이용해하드웨어로알고리즘을구현하는방법을설명합니다. Leros 의초기구현에도동일한접근방식을사용하겠습니다.

우리는모든명령어의데이터흐름을허용하는데이터패스가필요하며, 이는여러클록사이클에걸쳐이루어질수있습니다. 우리는각명령어를두클록사이클안에실행하는것을목표로합니다. 따라서기본상태기계는 fetch 와 execute 두상태를가집니다.

그림 14.1는 Leros 를구현하기위한데이터패스를보여줍니다. 그림은약간단순화되어있습니다. 데이터는왼쪽에서오른쪽으로흐릅니다. PC 는페치할명령어를가리킵니다. 온칩메모리는보통읽을수없는입력레지스터를가지고있습니다.[2] 따라서우리는 PC 와명령어메모리의입력레지스터에다음 PC 와같은값을공급합니다. 다음 PC 는분기하지않는명령어의경우 PC 에 1 을더한값입니다.[3] 상대분기의경우, 디코드컴포넌트가즉시값을부호확장하여 PC 에더합니다. jal 명령어의경우, PC 는 A 로부터로드될수도있습니다.

첫번째상태에서는명령어메모리로부터명령어를페치하고디코드합니다. 디코드컴포넌트는다음상태인실행상태에서무엇이일어날지를결정합니다. 디코드에는즉시피연산자를가진명령어 (예: addi 또는 loadhi) 를위한피연산자생성도포함됩니다. 이피연산자는실행상태에서소비되므로, 이를레지스터에저장해야합

[2] 적어도 FPGA 온칩메모리의경우에는그렇습니다

[3] 이단순한구성에서는바이트가아니라 16 비트명령어워드단위로계산합니다

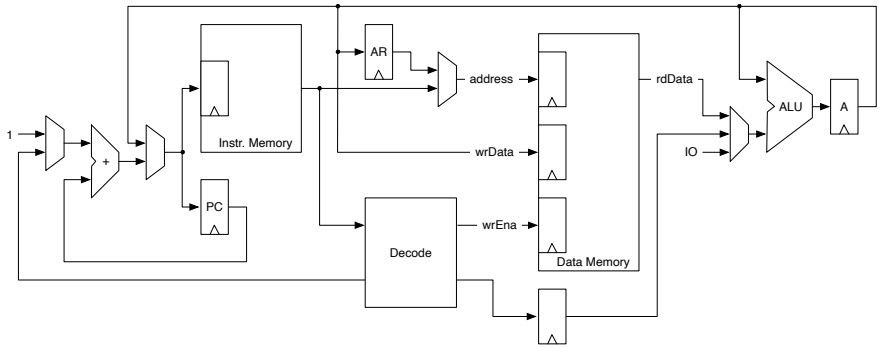


Figure 14.1: Leros 데이터패스.

니다.

두번째메모리는범용데이터메모리역할을하지만, 255 개레지스터의값도저장합니다. 레지스터중하나를읽거나쓸때, 주소는명령어의일부로주어집니다. 우리는메모리로드또는스토어명령어를위해주소레지스터 AR 을사용합니다. AR 자체는 A 로부터로드됩니다. 로드의결과는 A 에저장됩니다. 스토어시에는 A 가메모리에기록될데이터를전달합니다.

마지막으로, 산술및논리연산은 ALU 를이용해수행됩니다. 하나의피연산자는 A 에서오고, 다른하나는 (명령어로부터의) 즉시값이거나 (메모리로부터의) 레지스터값입니다.

14.3 ALU 로시작하기

프로세서의핵심컴포넌트는 산술논리장치, 줄여서 ALU 입니다. 따라서우리는 ALU 와테스트벤치를코딩하는것으로시작하겠습니다. 먼저, ALU 의다양한연산을나타내는상수를정의하겠습니다.

```
// Alu ops
val nop = 0
val add = 1
val sub = 2
val and = 3
val or = 4
val xor = 5
```

```
  val ld = 6
  val shr = 7
```

ALU 는보통두개의피연산자입력 (a 와 b 라고부르겠습니다), 함수를선택하기위한연산 op(또는오피코드) 입력, 그리고출력 y 를가집니다. 목록 14.2는 ALU 를보여줍니다.

TODO: 위키백과를참고하여멋진 *ALU* 를그려보십시오.

먼저세입력에대해더짧은이름을정의하겠습니다. switch 문은 res 의계산을위한로직을정의합니다. 따라서 res 는기본값 0 을할당받습니다. switch 문은모든연산을나열하고그에따라표현식을할당합니다. 모든연산은 Chisel 표현식으로직접매핑됩니다. 마지막으로, 우리는결과 res 를 ALU 출력 y 에할당합니다.

```
class AluAccu(size: Int) extends Module {
  val io = IO(new Bundle {
    val op = Input(UInt(3.W))
    val din = Input(UInt(size.W))
    val enaMask = Input(UInt(4.W))
    val enaByte = Input(Bool())
    val enaHalf = Input(Bool())
    val off = Input(UInt(2.W))
    val accu = Output(UInt(size.W))
  })

  val accuReg = RegInit(0.U(size.W))

  val op = io.op
  val a = accuReg
  val b = io.din
  val res = WireDefault(a)

  switch(op) {
    is(nop.U) {
      res := a
    }
    is(add.U) {
      res := a + b
    }
    is(sub.U) {
      res := a - b
    }
```

```
    is(and.U) {
      res := a & b
    }
    is(or.U) {
      res := a | b
    }
    is(xor.U) {
      res := a ^ b
    }
    is(shr.U) {
      res := a >> 1
    }
    is(ld.U) {
      res := b
    }
  }

  val byte = WireDefault(res(7, 0))
  val half = WireDefault(res(15, 0))
  when(io.off === 1.U) {
    byte := res(15, 8)
  }.elsewhen(io.off === 2.U) {
    byte := res(23, 16)
    half := res(31, 16)
  }.elsewhen(io.off === 3.U) {
    byte := res(31, 24)
  }
  val signExt = Wire(SInt(32.W))
  when (io.enaByte) {
    signExt := byte.asSInt
  } .otherwise {
    signExt := half.asSInt
  }

  // Workaround for missing subword assignments
  val split = Wire(Vec(4, UInt(8.W)))
  for (i <- 0 until 4) {
    split(i) := Mux(io.enaMask(i), res(8 * i + 7, 8 * i),
        accuReg(8 * i + 7, 8 * i))
  }
```

```
    def alu(a: Int, b: Int, op: Int): Int = {

      op match {
        case 0 => a
        case 1 => a + b
        case 2 => a - b
        case 3 => a & b
        case 4 => a | b
        case 5 => a ^ b
        case 6 => b
        case 7 => a >>> 1
        case _ => -123 // This shall not happen
      }
    }
```

Listing 14.3: Scala 로작성된 Leros ALU 함수입니다.

```
  when((io.enaByte || io.enaHalf) & io.enaMask.andR) {
    accuReg := signExt.asUInt
  } .otherwise {
    accuReg := split.asUInt
  }

  io.accu := accuReg
}
```

Listing 14.2: 누산레지스터를갖춘 Leros ALU 입니다.

테스트를위해, 목록 14.3에나타난것처럼 ALU 함수를순수 Scala 로작성합니다.
Chisel 로작성된하드웨어와 Scala 구현이이렇게중복되어도명세상의오류를탐지하지는못하지만, 적어도어느정도의정합성점검은됩니다. 테스트벡터로몇가지경계사례값을사용합니다.

```
// Some interesting corner cases
val interesting = Seq(1, 2, 4, 123, 0, -1, -2,
    0x80000000, 0x7fffffff)
```

입력한쌍을테스트하기위해함수 (testOne()) 를정의합니다.

```
def testOne(a: Int, b: Int, fun: Int): Unit = {
  dut.io.op.poke(ld.U)
  dut.io.enaMask.poke("b1111".U)
  dut.io.din.poke((a.toLong & 0x00ffffffL).U)
  dut.clock.step(1)
  dut.io.op.poke(fun.U)
  dut.io.din.poke((b.toLong & 0x00ffffffL).U)
  dut.clock.step(1)
  dut.io.accu.expect((alu(a, b, fun.toInt).toLong &
      0x00ffffffL).U)
}
```

그런다음두입력모두에대해그값들로모든함수를테스트합니다.

```
def test(values: Seq[Int]) = {
  for (fun <- 0 to 7) {
    for (a <- values) {
      for (b <- values) {
        testOne(a, b, fun)
      }
    }
  }
}
```

32 비트인자에대한완전한전수테스트는불가능하며, 이때문에입력값으로몇가지경계사례를선택하였습니다. 경계사례에대한테스트외에도, 무작위입력에대한테스트역시유용합니다.

```
val randArgs =
    Seq.fill(10)(scala.util.Random.nextInt())
test(randArgs)
```

Leros 프로젝트내에서다음명령으로테스트를실행할수있습니다

```
$ sbt "testOnly leros.AluAccuTest"
```

테스트는다음과유사한성공메시지를출력해야합니다.

```
[info] AluAccuTest:
[info] AluAccu
[info] - should pass
[info] Run completed in 1 second, 794 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
```

14.4 명령어디코딩

ALU 로부터거꾸로작업하여명령어디코더를구현합니다. 명령어디코딩이란다음단계/상태에서멀티플렉서와 ALU 를구동할신호를생성하는것입니다. 하지만먼저, 명령어인코딩을별도의 Scala 클래스와 공유패키지에정의합니다. 인코딩상수를 Leros 의하드웨어구현, Leros 용어셈블러, 명령어집합시뮬레이터사이에서공유하고자합니다.

```
object Constants {
  val NOP = 0x00
  val ADD = 0x08
  val ADDI = 0x09
  val SUB = 0x0c
  val SUBI = 0x0d
  val SHR = 0x10
  val LD = 0x20
  val LDI = 0x21
  val AND = 0x22
  val ANDI = 0x23
  val OR = 0x24
  val ORI = 0x25
  val XOR = 0x26
  val XORI = 0x27
  val LDHI = 0x29
  val LDH2I = 0x2a
  val LDH3I = 0x2b
  val ST = 0x30
  // ...
```

TODO: Leros 가더완성되면코드를갱신하십시오. 아직빠진내용이있기때문입니다.
디코드컴포넌트를위해출력용 Bundle 을정의하며, 이번들은나중에실행상태에서사용되고일부는 ALU 에입력됩니다. DecodeOut Bundle 은여기서보여주지않는더많은필드를포함하고있습니다. 자세한내용은원본 Leros 코드베이스를참조하십시오.

```
class DecodeOut extends Bundle {
  val operand = UInt(32.W)
  val enaMask = UInt(4.W)
  val op = UInt()
  val off = SInt(10.W)
  val isRegOpd = Bool()
  val useDecOpd = Bool()
  val isStore = Bool()
  // ... and more fields
```

또한 DecodeOut 클래스의동반객체 (companion object) 를정의하는데, 여기에는 DecodeOut 객체를생성하고모든필드를기본값으로설정하는함수 default() 가포함되어있습니다.

```
object DecodeOut {

  val MaskNone = "b0000".U
  val MaskAll = "b1111".U

  def default: DecodeOut = {
    val v = Wire(new DecodeOut)
    v.operand := 0.U
    v.enaMask := MaskNone
    v.op := nop.U
    v.off := 0.S
    v.isRegOpd := false.B
    v.useDecOpd := false.B
    v.isStore := false.B
    // ... and more fields
```

디코드는 8 비트오피코드를입력으로받아디코딩된신호를출력으로전달합니다. 이구동신호들은해당객체를생성하는 default 함수를사용하여기본값이할당됩니다.

```
class Decode() extends Module {
  val io = IO(new Bundle {
    val din = Input(UInt(16.W))
    val dout = Output(new DecodeOut)
  })

  import DecodeOut._

  val d = DecodeOut.default
```

디코딩자체는오피코드를나타내는명령어부분 (Leros 에서는대부분의명령어에 대해상위 8 비트) 에대한큰 switch 문에불과합니다.

```
  switch(instr(15, 8)) {
    is(ADD.U) {
      d.op := add.U
      d.enaMask := MaskAll
      d.isRegOpd := true.B
    }
    is(ADDI.U) {
      d.op := add.U
      d.enaMask := MaskAll
      d.useDecOpd := true.B
    }
    is(SUB.U) {
      d.op := sub.U
      d.enaMask := MaskAll
      d.isRegOpd := true.B
    }
    is(SUBI.U) {
      d.op := sub.U
      d.enaMask := MaskAll
      d.useDecOpd := true.B
    }
    is(SHR.U) {
      d.op := shr.U
      d.enaMask := MaskAll
    }
    // ...
```

또한디코드모듈은명령어내상수의부호확장버전도생성하고, 간접로드및저장

명령어를위한오프셋도계산합니다.

TODO: 디코딩에대해더다룰내용이있습니다 *TODO:* 그리고실행전체가빠져있습니다

14.5 명령어어셈블

Leros 용프로그램을작성하려면어셈블러가필요합니다. 하지만아주초기테스트를위해서는몇개의명령어를하드코딩하여 Scala 배열에담고, 이를사용해명령어메모리를초기화할수있습니다.

```
val prog = Array[Int](
  0x0903, // addi 0x3
  0x09ff, // -1
  0x0d02, // subi 2
  0x21ab, // ldi 0xab
  0x230f, // and 0x0f
  0x25c3, // or 0xc3
  0x0000
)

def getProgramFix() = prog
```

하지만이는프로세서를테스트하는데매우비효율적인접근방식입니다. Scala 처럼표현력이뛰어난언어로어셈블러를작성하는일은큰작업이아닙니다. 따라서우리는약 100 줄의코드로작성가능한간단한 Leros 용어셈블러를작성합니다. 어셈블러를호출하는 getProgram 함수를정의합니다. 분기목적지를위한심볼테이블이필요한데, 이는 Map 에수집합니다. 고전적인어셈블러는두번의패스로동작합니다: (1) 심볼테이블을위한값들을수집하고, (2) 첫번째패스에서수집한심볼을사용해프로그램을어셈블합니다. 따라서어느패스인지를나타내는매개변수와함께 assemble 을두번호출합니다.

```
def getProgram(prog: String) = {
  assemble(prog)
}

// collect destination addresses in first pass
val symbols = collection.mutable.Map[String, Int]()

def assemble(prog: String): Array[Int] = {
```

```
    assemble(prog, false)
    assemble(prog, true)
  }
```

assemble 함수는소스파일을여는것으로시작하며, 가능한두가지피연산자를파싱하기위한두개의헬퍼함수를정의합니다: (1) 정수상수 (십진수또는십육진수 표기허용) 그리고 (2) 레지스터번호를읽는것입니다.

```
  def assemble(prog: String, pass2: Boolean): Array[Int] = {

    val source = Source.fromFile(prog)
    var program = List[Int]()
    var pc = 0

    def toInt(s: String): Int = {
      if (s.startsWith("0x")) {
        Integer.parseInt(s.substring(2), 16)
      } else {
        Integer.parseInt(s)
      }
    }

    def regNumber(s: String): Int = {
      assert(s.startsWith("r"), "Register numbers shall
          start with \'r\'")
      s.substring(1).toInt
    }
```

Listing 14.4는 Leros 어셈블러의핵심을보여줍니다. Scala match 표현식이어셈블함수의핵심을담당합니다. *TODO:* 이코드에대해몇마디더하겠습니다.

14.6 명령어메모리

Listing 14.5는 Leros 의명령어메모리모듈을보여줍니다. 이메모리는주소비트수 (memAddrWidth) 를크기로, 그리고프로그램경로 (prog) 로설정됩니다. 명령어메모리의생성자는이전절에서소개한어셈블러를호출하여프로그램을어셈블합니다. 이는하드웨어생성과정에서임베디드프로세서를위한코드를어셈블하는하드웨어생성기의한예입니다. Leros 프로그램을담고있는 Scala 배열은 Scala Seq 로변환된다음, 익명함수 _.asUInt(16.W) 를통해 Chisel Vec 로매핑

```
for (line <- source.getLines()) {
  if (!pass2) println(line)
  val tokens = line.trim.split(" ")
  val Pattern = "(.*:)".r
  val instr = tokens(0) match {
    case "//" => // comment
    case Pattern(l) => if (!pass2) symbols +=
        (l.substring(0, l.length - 1) -> pc)
    case "add" => (ADD << 8) + regNumber(tokens(1))
    case "sub" => (SUB << 8) + regNumber(tokens(1))
    case "and" => (AND << 8) + regNumber(tokens(1))
    case "or" => (OR << 8) + regNumber(tokens(1))
    case "xor" => (XOR << 8) + regNumber(tokens(1))
    case "load" => (LD << 8) + regNumber(tokens(1))
    case "addi" => (ADDI << 8) + toInt(tokens(1))
    case "subi" => (SUBI << 8) + toInt(tokens(1))
    case "andi" => (ANDI << 8) + toInt(tokens(1))
    case "ori" => (ORI << 8) + toInt(tokens(1))
    case "xori" => (XORI << 8) + toInt(tokens(1))
    case "shr" => (SHR << 8)
    // ...
    case "" => // println("Empty line")
    case t: String => throw new Exception("Assembler
        error: unknown instruction: " + t)
    case _ => throw new Exception("Assembler error")
  }
```

Listing 14.4: Leros 어셈블러의주요부분입니다.

```
class InstrMem(memAddrWidth: Int, prog: String) extends
    Module {
  val io = IO(new Bundle {
    val addr = Input(UInt(memAddrWidth.W))
    val instr = Output(UInt(16.W))
  })
  val code = Assembler.getProgram(prog)
  assert(scala.math.pow(2, memAddrWidth) >= code.length,
      "Program too large")
  val progMem =
      VecInit(code.toIndexedSeq.map(_.asUInt(16.W)))
  val memReg = RegInit(0.U(memAddrWidth.W))
  memReg := io.addr
  io.instr := progMem(memReg)
}
```

Listing 14.5: Leros 의명령어메모리입니다.

됩니다. 명령어메모리는 FPGA 에서온칩메모리로구현될수있도록주소용레지스터 (memReg) 를포함합니다.[4]

14.7 데이터패스를갖춘상태기계구현

이장의시작부분에서언급했듯이, Leros ISA 정의는구체적인구현을정의하지않습니다. 이장전체에걸쳐우리는암묵적인설계결정들을내려왔습니다. 여기서는하나의설계옵션에대해논의하겠습니다.

제시된구현에서는데이터메모리를레지스터파일과공유합니다. 256 개의레지스터는단지데이터메모리내의배열일뿐입니다. 다른구현에서는데이터와레지스터를위한전용온칩메모리를둘수도있습니다.

우리는 Leros 를데이터패스를갖춘상태기계라는아이디어로구현했습니다. 이는곧각명령어가하나이상의클록사이클을소요한다는것을의미합니다. Listing 14.6에서보듯이우리는 fetch 와 execute 라는두가지상태를사용합니다.

[4] 현재버전의 Chisel 에서는생성된코드에 FPGA 합성도구가온칩메모리로매핑할수없는큰우선순위 Mux 가포함됩니다. MLIR 백엔드를사용하면이문제가해결될것입니다. 이문제에대한또다른우회방법으로는, Patmos 프로젝트의부트로더에서했던것처럼, 합성도구에적합한 Verilog 코드를생성하여블랙박스로포함하는방법이있습니다.

```
object State extends ChiselEnum {
  val fetch, execute = Value
}
import State._

val stateReg = RegInit(fetch)

switch(stateReg) {
  is(fetch) {
    stateReg := execute
  }
  is(execute) {
    stateReg := fetch
  }
}
```

Listing 14.6: Leros 의상태기계입니다.

상태기계는이두상태사이를전환합니다. fetch 상태에서는명령어메모리로부터명령어를페치하고, 해당명령어를디코딩하기도합니다. 또한데이터메모리가동기식이며읽기결과를전달하는데한클록사이클이필요하므로, fetch 상태에서데이터메모리로부터의읽기연산도시작합니다.

execute 상태에서는누산기에대한새로운값을계산하거나, 데이터메모리로부터읽은결과를누산기에저장합니다. 또한 execute 상태에서쓰기작업도수행합니다.

다음코드는누산기와두개의주요상태레지스터인프로그램카운터 (pcReg) 와주소레지스터 (addrReg) 를포함하여 ALU 의인스턴스화를보여줍니다.

```
val alu = Module(new AluAccu(size))
val accu = alu.io.accu

// The main architectural state
val pcReg = RegInit(0.U(memAddrWidth.W))
val addrReg = RegInit(0.U(memAddrWidth.W))

val pcNext = WireDefault(pcReg + 1.U)
```

다음코드는명령어메모리의인스턴스화를보여줍니다. 명령어메모리는프로그

램의파일이름을매개변수로가진다는점에유의하십시오.

```
val mem = Module(new InstrMem(memAddrWidth, prog))
mem.io.addr := pcNext
val instr = mem.io.instr
```

다음코드는디코드모듈의인스턴스화를보여줍니다. 디코드모듈의입력은페치모듈에서온명령어이며, 출력은디코드신호입니다. 이신호들은 execute 상태에서필요하므로 decReg 에레지스터로저장됩니다.

```
val dec = Module(new Decode())
dec.io.din := instr
val decout = dec.io.dout

val decReg = RegInit(DecodeOut.default)
when (stateReg === fetch) {
  decReg := decout
}
```

Listing 14.7은 Leros 의데이터메모리를보여줍니다. 메모리는 32 비트워드로구성됩니다. 이러한 32 비트워드에바이트단위접근을가능하게하기위해, 워드는네개의 8 비트바이트로이루어진 Vec 으로분할됩니다. read() 연산은네바이트로이루어진벡터를반환하며, 이를 ## 연산자로연결하여 32 비트워드로만듭니다. 쓰기의경우, 쓰려는워드를이네바이트로분할하고쓰기마스크 (wrMask)를사용하여어떤바이트가쓰일지선택합니다. SyncReadMem 컴포넌트에는벡터와쓰기마스크를매개변수로받는 write() 함수가포함되어있습니다.

다음코드는데이터메모리의인스턴스화와포트연결을보여줍니다.

```
val dataMem = Module(new DataMem((memAddrWidth)))

val memAddr = Mux(decout.isDataAccess, effAddrWord,
    instr(7, 0))
val memAddrReg = RegNext(memAddr)
val effAddrOffReg = RegNext(effAddrOff)
dataMem.io.rdAddr := memAddr
val dataRead = dataMem.io.rdData
dataMem.io.wrAddr := memAddrReg
dataMem.io.wrData := accu
dataMem.io.wr := false.B
dataMem.io.wrMask := "b1111".U
```

```
class DataMem(memAddrWidth: Int) extends Module {
  val io = IO(new Bundle {
    val rdAddr = Input(UInt(memAddrWidth.W))
    val rdData = Output(UInt(32.W))
    val wrAddr = Input(UInt(memAddrWidth.W))
    val wrData = Input(UInt(32.W))
    val wr = Input(Bool())
    val wrMask = Input(UInt(4.W))
  })

  val mem = SyncReadMem(1 << memAddrWidth, Vec(4, UInt(8.W)))

  val rdVec = mem.read(io.rdAddr)
  io.rdData := rdVec(3) ## rdVec(2) ## rdVec(1) ## rdVec(0)
  val wrVec = Wire(Vec(4, UInt(8.W)))
  val wrMask = Wire(Vec(4, Bool()))
  for (i <- 0 until 4) {
    wrVec(i) := io.wrData(i * 8 + 7, i * 8)
    wrMask(i) := io.wrMask(i)
  }
  when (io.wr) {
    mem.write(io.wrAddr, wrVec, wrMask)
  }
}
```

Listing 14.7: Leros 의데이터메모리모듈입니다.

14.8 구현변형

실제프로세서는 명령어파이프라이닝을수행합니다. 명령어파이프라인에서는 하나이상의명령어가동시에진행중입니다. 예를들어 Leros 의경우, 명령어페치, 명령어디코드, 실행이라는세개의파이프라인단계를구현할수있습니다. 이경우세개의명령어가파이프라인에존재하게되며, 매클록사이클마다하나의명령어를실행할수있습니다. 지금까지제시한구현과비교하면, 파이프라이닝은 Leros 의성능을약두배로향상시킬수있습니다. 다음장에서는 RISC-V 명령어집합을구현하는파이프라인프로세서를소개하겠습니다.

14.9 연습문제

마지막몇장중하나에해당하는이연습과제는매우자유로운형식입니다. 여러분은 Chisel 을통한학습여정의끝에도달했으며, 흥미롭다고느끼는설계문제들을다룰준비가되어있습니다.

한가지선택지는이장을다시읽으면서 Leros 저장소에있는모든소스코드를함께살펴보고, 테스트케이스를실행하고, 코드를일부러망가뜨려보면서테스트가실패하는모습을확인하는것입니다.

또다른선택지는여러분스스로 Leros 의구현을작성해보는것입니다. 저장소에있는구현은가능한여러구성방식중하나일뿐입니다. 단일파이프라인단계를가진 Leros 의 Chisel 시뮬레이션버전을작성해볼수도있고, 과감하게최고의클록주파수를목표로 Leros 를슈퍼파이프라인화해볼수도있습니다.

세번째선택지는여러분자신의프로세서를처음부터설계하는것입니다. Leros 프로세서를구축하는방법과필요한도구들에대한이시연이, 프로세서설계와구현이마법같은기술이아니라매우즐거운엔지니어링이될수있음을여러분에게확신시켜주었을지도모릅니다.

15 RISC-V 파이프라인

파이프라이닝은어떤처리과정의처리량을향상시키는기법입니다. 예를들어, 자동차공장의조립라인에서는각단계가엔진장착과같은특화된작업을수행합니다. 그작업이끝나면자동차는다음단계로이동합니다. 이러한파이프라이닝은자동차제조라인의여러단계에서서로다른작업들을병렬로실행할수있게해줍니다.

디지털설계에서도파이프라이닝을활용하여데이터스트림처리속도를높일수있습니다. 디지털설계에서파이프라이닝의대표적인예는파이프라인마이크로프로세서를구축하는것입니다. 이경우명령어들이파이프라인을따라흐르면서, 각단계마다서로다른프로세서유닛이하나의연산을수행합니다. Chapter 14에서제시한, 각명령어가여러클록사이클을소요하는프로세서와달리, 파이프라인프로세서에서는명령어들이서로다른단계에서병렬로실행됩니다. 그결과로얻어지는 (이상적인) 처리량은클록사이클당하나의명령어입니다.

이장에서는간단한파이프라인 RISC-V 프로세서인 Wildcat [20] 을소개합니다. RISC-V 는원래캘리포니아대학교버클리캠퍼스에서개발된공개명령어집합구조입니다. Wildcat 은교육에사용할수있는가독성좋은 Chisel 코드를제공하는데초점을맞추고있습니다. 이장에는 RISC-V 파이프라인을구축하기위한가장중요한소스코드조각들이담겨있습니다. 더자세한내용과테스트는 Wildcat GitHub 저장소에서확인할수있습니다. 이저장소에는 Scala 로작성된 RISC-V 명령어집합시뮬레이터와, 시연을위한 Chisel 단일사이클버전도포함되어있습니다.

15.1 RISC-V 명령어집합구조

Andrew Waterman 은캘리포니아대학교버클리캠퍼스에서의박사학위논문 [30] 에서 RISC-V 명령어집합구조 (ISA) 를정의했습니다. Krste Asanovic 과 Dave Patterson 이그의지도교수였습니다. Waterman 은 MIPS, SPARC, Alpha 를비롯한지난삼십년간의 RISC 아키텍처들을탐구하여 RISC 의정수를 RISC-V ISA 정의로응축해냈습니다. RISC-V ISA 는오픈소스로제공됩니다. 이때문에많은마이크로컨트롤러제공업체들이지난몇년사이 RISC-V 로전환했습니다.

RISC-V 는하나의 ISA 정의이며, 구현을정의하지는않습니다. "V" 는캘리포니아대학교버클리캠퍼스에서진행된다섯번째 RISC 프로젝트를나타내며, 동시에벡터명령어가이표준의일부임을나타내기도합니다.

RISC-V ISA 정의는 32 비트및 64 비트아키텍처를위한정수명령어집합에대해 RV32I 와 RV64I 라는두가지기본버전을정의합니다. 곱셈과나눗셈을위해 ISA 를확장하는 M 과같은선택적확장, 부동소수점을위한 F 와 D, 그리고그밖의여러확장이기본 ISA 를확장합니다. 이절에서는기본 RV32I 의구현을보여드리겠습니다.

RV32I ISA 는 32 비트크기의레지스터 32 개 (레지스터파일이라고도함) 를포함하는프로세서를정의하며, 여기서레지스터 X0 은항상 0 입니다. 또한, 프로세서상태의일부로서실행되는명령어를가리키는프로그램카운터 (PC) 가있습니다. 명령어는 32 비트폭입니다. 명령어와데이터는바이트단위로주소지정이가능한메모리에위치합니다. 프로세서상태는 31 개의레지스터와 PC 입니다.

RISC 아키텍처는로드-스토어아키텍처라고도불리는데, 이는모든피연산자가먼저메모리에서로드되어야하고, 연산 (예: 덧셈) 이후에는다시메모리에저장되어야하기때문입니다. 기본명령어집합은다음과같은유형의연산으로구성됩니다:

- 레지스터간의산술및논리연산으로, 결과는레지스터에기록됩니다. 예를들어: add x1, x2, x3 는레지스터 x2 와 x3 의내용을더하여그결과를레지스터 x1 에넣습니다.
- 레지스터와즉치값 (상수) 간의산술및논리연산으로, 결과는레지스터에기록됩니다. 예를들어: add x1, x2, 42 는레지스터 x2 의내용에 42 를더하여그결과를레지스터 x1 에넣습니다.
- 로드명령어는메모리에서값을읽어레지스터에로드합니다. 이명령어는레지스터의내용을기준주소로사용하고, 오프셋을더하여유효주소를계산합니다. 로드명령어의예는다음과같습니다: lw x3, 4(x1), 여기서기준주소는 x1 에있고오프셋은 4 바이트입니다. 결과는레지스터 x3 에저장됩니다. 바이트, 하프워드, 워드크기에대한로드명령어는부호없는확장버전과부호있는확장버전으로제공됩니다.
- 스토어명령어는레지스터의값을메모리에저장합니다. 주소계산방식은로드명령어와동일합니다. 예를들어: sw x2, 4(x1) 에서는레지스터 x2 의내용이주소 x1 + 4 의메모리에저장됩니다. 스토어명령어는바이트, 하프워드, 워드크기에대해제공됩니다.
- 제어명령어는 PC 를변경함으로써프로그램흐름을바꿀수있습니다. 조건부분기는조건 (레지스터간의비교) 을평가하여조건이충족되면분기합니

다. 분기오프셋은현재 PC 를기준으로상대적입니다. 분기명령어의예는다음과같습니다: bne x2, x3, fail. 점프명령어는무조건적으로 PC 를변경합니다. 점프명령어의한변형은함수호출로부터의복귀를가능하게하기위해다음명령어주소를레지스터에저장합니다. 이러한명령어의예는다음과같습니다: jal x1, foo, 여기서프로세서는무조건적으로함수 foo 로점프하고복귀주소를레지스터 x1 에저장합니다.

RISC-V ISA 에대한상세한설명은 Patterson 과 Hennessy 의고전적인교과서 [14] 를읽거나공식 RISC-V Instruction Set Manual을참고하십시오.

15.2 파이프라인단계정의

파이프라인단계는하나의클록사이클내에서특정기능을수행합니다. 그기능은조합함수입니다. 파이프라인단계사이에는중간결과를저장하기위해레지스터가사용됩니다.

레지스터가이러한단계들사이에위치하므로, 어느레지스터가해당단계에속하는지정의해야합니다: 입력레지스터인지출력레지스터인지입니다. 실용적인이유로, 사용가능한온칩메모리를고려하여입력레지스터를파이프라인단계의일부로간주하겠습니다.

15.3 파이프라인단계의수

아키텍처는파이프라인단계의수를정의합니다. 파이프라인이길어지면설계를더높은주파수로클로킹할수있습니다. 그러나파이프라인단계가추가될때마다레지스터로인한오버헤드가증가하며, 설계는더복잡해집니다.

축소명령어집합컴퓨터(RISC) 의고전적인구성은 5 단계파이프라인입니다:

1. 명령어인출
2. 명령어해독및레지스터파일읽기
3. 실행
4. 메모리접근
5. 결과기록

이러한구성은많은컴퓨터구조교과서, 예를들어 [14] 에서사용됩니다. 그러나 RISC-V 파이프라인설계는 2 단계부터여러단계이상까지존재합니다.

파이프라인단계의수를결정하는한가지요소는명령어및데이터스크래치패드메모리또는명령어및데이터캐시로사용가능한온칩메모리입니다. 현재의온칩메모리는입력 (주소및쓰기데이터) 에레지스터를가지고있습니다. 제 6.4절을참고하십시오. 그입력레지스터는파이프라인의일부이며, 곧파이프라인레지스터입니다. 메모리읽기는한클록사이클의지연시간을갖습니다. 설계를단순화하기위해, 3 단계 RISC-V 파이프라인을제시합니다:

1. 명령어인출
2. 명령어해독, 레지스터파일읽기, 그리고주소계산
3. 실행및메모리접근

15.4 Wildcat 파이프라인

그림 15.1는 3 단 Wildcat 파이프라인을보여줍니다. 단순화를위해, 명령어메모리 (IM), 레지스터파일 (RF), 데이터메모리 (DM) 에는온칩메모리가사용된다고가정합니다. 이세온칩메모리가단계수의하한을 3 으로설정합니다.[1]

명령어는왼쪽에서오른쪽으로흐릅니다. 명령어의결과는레지스터파일 (RF) 이나데이터메모리 (DM) 에오른쪽에서왼쪽으로다시기록됩니다.

15.4.1 최상위레벨

리스트 15.1는 wildcat 구현들을위한최상위레벨모듈의추상상위클래스를보여줍니다.[2] 이인터페이스는명령어메모리와데이터메모리로의연결만으로구성됩니다. 어떤메모리를사용할지에대해서는유연하게대처하는데, 예를들어소규모구현을위한스크래치패드메모리나캐시를사용할수있습니다. IO 장치는데이터메모리와멀티플렉싱됩니다. 메모리와 IO 의정의는시스템온칩 (SoC) 의최상위레벨에서이루어져야합니다.

[1] RF 를구체적인플립플롭으로구성한다면, 그러한 RF 는비동기적으로읽을수있으므로단계수를 2 로줄일수있습니다.

[2] 이인터페이스는여러구현변형, 예컨대서로다른파이프라인구성들과공유됩니다.

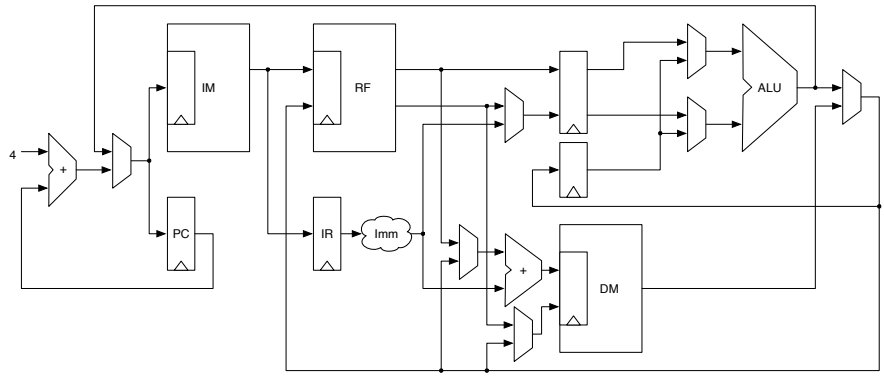


Figure 15.1: 3 단 Wildcat 프로세서파이프라인 (단순화된형태이며, 제어신호와디코딩된신호는생략함).

```
abstract class Wildcat() extends Module {
  val io = IO(new Bundle {
    val imem = new InstrIO()
    val dmem = new MemIO()
  })
}
```

Listing 15.1: Wildcat 의최상위레벨모듈.

```
// PC generation
// val pcReg = RegInit(-4.S(32.W).asUInt)
val pcReg = RegInit(0.S(32.W).asUInt)
val pcNext = WireDefault(Mux(doBranch, branchTarget, pcReg
    + 4.U))
pcReg := pcNext
io.imem.address := pcNext

// Fetch
val instr = WireDefault(io.imem.data)
when (io.imem.stall) {
  instr := 0x00000013.U
  pcNext := pcReg
}
```

Listing 15.2: PC 생성과명령어페치.

15.4.2 명령어페치

프로그램카운터 (PC) 는실행될다음명령어를가리킵니다. RISC-V 는 32 비트폭명령어를가지므로, PC 는순차적인명령어마다 4 씩증가합니다. 분기명령어의경우 PC 는그에맞게설정되며, 이는가산기앞의멀티플렉서로나타나있습니다.

그림에서볼수있듯이, IM 은입력레지스터를포함하고있으며, 이는페치단계의파이프라인레지스터의일부입니다. 그러나 PC 역시파이프라인레지스터의일부입니다. 따라서 PC 레지스터의출력을 IM 에공급할수는없으며, PC 의 다음값을공급해야합니다. IM 과 PC 의주소입력은항상동일한데이터를담고있습니다.

리스트 15.2는페치단계의코드를보여줍니다. PC(pcReg) 는 -4 로초기화되어, 리셋후 pcNext 의값이 0 이되도록합니다. doBranch 신호는분기대상과현재 PC 에 4 를더한값사이에서선택합니다. pcNext 신호는명령어메모리의입력에연결됩니다. 명령어메모리의출력은 instr 에연결됩니다. 파이프라인을스톨해야하는경우 (예를들어캐시미스의경우), NOP 가명령어를대체하며, pcNext 는증가하지않습니다.

시뮬레이션과 FPGA 에서의소규모실험을위해, 하드웨어생성시점에미리로드되는읽기전용메모리 (ROM) 를사용합니다. 리스트 15.3는그메모리를보여줍니다. 이는 Scala Array 의내용으로초기화되는 Vec 을사용합니다. ROM 은

```
class InstructionROM(code: Array[Int]) extends Module {
  val io = IO(Flipped(new InstrIO()))

  val addrReg = RegInit(0.U(32.W))
  addrReg := io.address
  val instructions =
      VecInit(code.toIndexedSeq.map(_.S(32.W).asUInt))
  io.data := instructions(addrReg(31, 2))
  io.stall := false.B
}
```

Listing 15.3: 간단한명령어메모리로서의 ROM.

주소를위한레지스터를포함하고있으며, 이는파이프라인레지스터의일부입니다.

15.4.3 명령어디코드와레지스터파일읽기

두번째파이프라인단계는명령어의디코딩, 레지스터파일읽기, 그리고메모리연산을위한주소계산을수행합니다. 파이프라인레지스터는명령어레지스터와레지스터파일을위한주소레지스터로구성됩니다. RISC-V ISA 인코딩에서, 레지스터주소를담고있는필드는항상명령어내동일한위치에있습니다. 따라서명령어가아직디코딩되지않았더라도레지스터파일을읽을수있습니다. 이값들이필요하지않은경우에는그저무시됩니다.

리스트 15.4는디코드단계의파이프라인레지스터 (instrReg) 와레지스터파일의입력이되는 rs1, rs2 신호를보여줍니다. 레지스터파일자체와디코드하드웨어는모듈보다더경량화된형태인함수로구현됩니다.

레지스터파일구현의핵심은 Listing 15.5에나와있습니다. 가능하다면온칩메모리에레지스터파일을구현하기위해 SyncReadMem 을사용합니다.[3] Section 6.4에나온것처럼명시적으로처리하는대신, 같은클록사이클에동일한레지스터를쓰고읽을때포워딩을가능하게하기위해 SyncReadMem.WriteFirst 매개변수를사용한다는점에유의하십시오.

FPGA 에서는온칩메모리가구성시점에 0 으로초기화됩니다. 그러나 ASIC 에서는메모리의내용이정의되지않습니다. 따라서레지스터 0 에서읽을때온칩메모리의무작위내용이아니라 0 을반환하도록특수한경우를처리해야합니다.

[3] 메모리컴파일러를사용하지않고, 예를들어오픈소스 OpenLane 플로우로 ASIC 용메모리를합성하는경우, 메모리는전용플립플롭으로구현됩니다.

```
  val instrReg = RegInit(0x00000033.U) // nop on reset
  instrReg := Mux(doBranch, 0x00000033.U, instr)
  val rs1 = instr(19, 15)
  val rs2 = instr(24, 20)
  val rd = instr(11, 7)
  val (rs1Val, rs2Val, debugRegs) = registerFile(rs1, rs2,
      wbDest, wbData, wrEna, true)

  val decOut = decode(instrReg)
```

Listing 15.4: 디코드단계.

```
    val regs = SyncReadMem(32, UInt(32.W),
        SyncReadMem.WriteFirst)
    val rs1Val = Mux(RegNext(rs1) === 0.U, 0.U,
        regs.read(rs1))
    val rs2Val = Mux(RegNext(rs2) === 0.U, 0.U,
        regs.read(rs2))
    when(wrEna && rd =/= 0.U) {
      regs.write(rd, wrData)
    }
    (rs1Val, rs2Val, debugRegs)
```

Listing 15.5: 레지스터파일함수의핵심.

read() 함수는한클록사이클의지연시간을두고결과를반환한다는점에유의하십시오. 따라서메모리출력의멀티플렉서는한클록사이클지연된레지스터주소와비교되어야합니다. (이는 Section 6.4에나온것처럼, 쓰기중읽기에대한수동포워딩과유사합니다.) 이함수는두개의읽은레지스터값을반환하는 Scala 튜플을반환합니다.[4]

레지스터파일은두개의읽기포트를가지고있는데, 이는 FPGA 에서흔하지않습니다. 따라서실제구현에서는동일한데이터로기록되는두개의온칩메모리를사용하여, 각각이하나의읽기포트를제공하도록합니다.

리스트 15.6는디코드함수의일부를보여줍니다. RISC-V 명령어를디코딩하기위해서는명령어의 opcode 필드와 func3 필드를읽어야합니다. decOut 은디코

[4] 이함수는디버깅레지스터도반환하지만, 합성시에는무시됩니다.

딩의출력입니다. 초기단계로서, 모든필드에기본값을할당합니다. 그뒤를이어 opcode 를명령어유형을나타내는상수들과비교하는큰 switch 문이따라옵니다. 이상수들은 Scala 정수로정의되어, Wildcat 의하드웨어구현과 Scala 로작성된명령어집합시뮬레이터사이에서공유될수있습니다. 따라서이상수들은 .U 메서드를사용하여 Chisel 상수로변환됩니다.

유사한함수가명령어로부터 ALU 연산을디코딩합니다. 또한명령어로부터가능한즉값이추출됩니다. 이모든값은디코딩된출력의일부입니다.

리스트 15.7는메모리로드또는저장명령어에대한주소계산을보여줍니다. 주소 (memAdress) 는레지스터 rs1 의값을사용하고즉값필드 (dec.Out.imm) 를더하여계산됩니다. 그러나포워딩을구현해야하므로코드는약간더복잡합니다. 포워딩은직전명령어들에서방금기록된레지스터를주소로사용할경우필요합니다. 이경우, 값은아직레지스터파일에다시기록되지않았으므로실행단계로부터 포워드합니다. 이코드스니펫은또한메모리저장연산에대한기록데이터가어디서오는지도보여줍니다. 레지스터 rs2 의값이사용되거나, 실행단계로부터값이포워딩됩니다.

리스트 15.8은디코드단계에서의메모리접근부분을보여줍니다. 메모리의입력레지스터가실행단계의일부이므로, 메모리주소와저장연산은디코드단계에서계산됩니다.

15.4.4 실행및메모리읽기

세번째파이프라인단계는 ALU 연산을실행하거나, 제어명령어 (분기또는점프)를실행하거나, 메모리로드를수행하는역할을담당합니다. 리스트 15.9는실행단계의일부를보여줍니다. 명령어유형에따라, ALU 명령어의두번째피연산자로레지스터 v2 의값또는즉값 decOut.imm 이사용됩니다. ALU 는디코드함수와마찬가지로 Scala 함수로정의됩니다. 이코드스니펫은또한상위즉값을로드하고상위즉값을 PC 에더하는구현도보여줍니다.

리스트 15.10에서보듯이, ALU 연산을정의하기위해 Scala Enumeration 을사용합니다. 다른상수들과마찬가지로, 이들은 ISA 시뮬레이터와파이프라인구현사이에서공유됩니다. 따라서 Chisel 상수로변환되어야합니다.

리스트 15.11는 ALU 연산을구현하는함수를보여줍니다.

리스트 15.12는분기실행에대한로직을보여줍니다. 분기대상은분기명령어의즉값을해당명령어의 PC 에더하거나, 간접분기의경우레지스터값을사용하여계산됩니다. ALU 와마찬가지로, 조건부분기를구현하는데필요한비교연산을위해함수 (compare()) 를사용합니다.

메모리읽기연산은실행단계에서수행됩니다. 리스트 15.13는로드명령어를위한멀티플렉서를보여줍니다. 함수 selectLoadData 는명령어와주소의하위 2

```
def decode(instruction: UInt) = {

  val opcode = instruction(6, 0)
  val func3 = instruction(14, 12)
  val decOut = Wire(new DecodedInstr())
  decOut.instrType := R.id.U
  decOut.isImm := false.B
  decOut.isLui := false.B
  decOut.isAuiPc := false.B
  decOut.isLoad := false.B
  decOut.isStore := false.B
  decOut.isBranch := false.B
  decOut.isJal := false.B
  decOut.isJalr := false.B
  decOut.rfWrite := false.B
  decOut.isECall := false.B
  decOut.isCssrw := false.B
  decOut.rs1Valid := false.B
  decOut.rs2Valid := false.B
  switch(opcode) {
    is(AluImm.U) {
      decOut.instrType := I.id.U
      decOut.isImm := true.B
      decOut.rfWrite := true.B
      decOut.rs1Valid := true.B
    }
    is(Alu.U) {
      decOut.instrType := R.id.U
      decOut.rfWrite := true.B
      decOut.rs1Valid := true.B
      decOut.rs2Valid := true.B
    }
    // and more cases
```

Listing 15.6: 레지스터파일함수의핵심.

```
// Forwarding to memory
val address = Mux(wrEna && (wbDest =/= 0.U) && wbDest ===
    decEx.rs1, wbData, rs1Val)
val data = Mux(wrEna && (wbDest =/= 0.U) && wbDest ===
    decEx.rs2, wbData, rs2Val)

val memAddress = (address.asSInt + decOut.imm).asUInt
decEx.memLow := memAddress(1, 0)
```

Listing 15.7: 필요한경우포워딩을포함한주소계산

```
io.dmem.rdAddress := memAddress
io.dmem.rdEnable := false.B
io.dmem.wrAddress := memAddress
io.dmem.wrData := data
io.dmem.wrEnable := VecInit(Seq.fill(4)(false.B))
when(decOut.isLoad && !doBranch) {
  io.dmem.rdEnable := true.B
}
when(decOut.isStore && !doBranch) {
  val (wrd, wre) = getWriteData(data, decEx.func3,
      memAddress(1, 0))
  io.dmem.wrData := wrd
  io.dmem.wrEnable := wre
}
```

Listing 15.8: 디코드단계에서의메모리접근

```
  val res = Wire(UInt(32.W))
  val val2 = Mux(decExReg.decOut.isImm,
      decExReg.decOut.imm.asUInt, v2)
  res := alu(decExReg.decOut.aluOp, v1, val2)
  when(decExReg.decOut.isLui) {
    res := decExReg.decOut.imm.asUInt
  }
  when(decExReg.decOut.isAuiPc) {
    res := (decExReg.pc.asSInt + decExReg.decOut.imm).asUInt
  }
```

Listing 15.9: 실행단계에서의 ALU 연산

```
object AluType extends Enumeration {
  type AluType = Value
  val ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND =
      Value
}
```

Listing 15.10: ALU 연산열거형

```
  def alu(op: UInt, a: UInt, b: UInt): UInt = {
    val res = Wire(UInt(32.W))
    res := DontCare
    switch(op) {
      is(ADD.id.U) {
        res := a + b
      }
      is(SUB.id.U) {
        res := a - b
      }
      is(AND.id.U) {
        res := a & b
      }
      is(OR.id.U) {
        res := a | b
      }
      is(XOR.id.U) {
        res := a ^ b
      }
      is(SLL.id.U) {
        res := a << b(4, 0)
      }
      is(SRL.id.U) {
        res := a >> b(4, 0)
      }
      is(SRA.id.U) {
        res := (a.asSInt >> b(4, 0)).asUInt
      }
      is(SLT.id.U) {
        res := (a.asSInt < b.asSInt).asUInt
      }
      is(SLTU.id.U) {
        res := (a < b).asUInt
      }
    }
    res
  }
```

Listing 15.11: ALU 함수

```
branchTarget := (decExReg.pc.asSInt +
    decExReg.decOut.imm).asUInt
when(decExReg.decOut.isJalr) {
  branchTarget := res
}
doBranch := ((compare(decExReg.func3, v1, v2) &&
    decExReg.decOut.isBranch) || decExReg.decOut.isJal ||
    decExReg.decOut.isJalr) && decExReg.valid
```

Listing 15.12: 분기실행

```
when(decExReg.decOut.isLoad && !doBranch) {
  res := selectLoadData(io.dmem.rdData, decExReg.func3,
      decExReg.memLow)
}
```

Listing 15.13: 메모리로드

비트에따라읽은데이터의어느부분이사용될지선택합니다.

15.5 요약및연습문제

이장에서는디지털설계에서파이프라이닝의사용법을보여주었습니다. 파이프라이닝을시연하기위해 RISC 프로세서설계를사용하였습니다. 이장에서는 Wildcat RISC-V 프로세서의가장중요한부분들을보여주었습니다. 테스트케이스를포함한전체소스코드는 Wildcat GitHub 저장소에서확인하십시오.

연습문제로서, 여기서보여드린코드조각들을바탕으로자신만의 RISC-V 프로세서버전을구현해볼수있습니다.

16 Chisel 에기여하기

Chisel 은지속적으로개발되고개선되고있는오픈소스프로젝트입니다. 따라서여러분도이프로젝트에기여할수있습니다. 여러분의기여는두가지방식일수있습니다: (1) 여러분의 Chisel 회로를오픈소스로그리고라이브러리로공개하는것, 또는 (2) Chisel 자체에개선사항을기여하는것입니다. 여기서는먼저라이브러리를공개하는방법을, 그다음 Chisel 라이브러리개발을위한환경을설정하는방법과 Chisel 에기여하는방법을설명합니다.

16.1 Chisel 라이브러리공개하기

오픈소스회로를개발하여예를들어 GitHub 에공유하는것은매우교육적인행위입니다. 다른사람들이여러분의코드예제를통해 Chisel 로하드웨어를기술하는법을배울수있기때문입니다. 그러나소스코드만공유하면다른사람들이그코드를자신의프로젝트에복사할수밖에없게됩니다. 이는적어도두가지문제를야기합니다: (1) 두개의사본은결코동일하지않습니다.[1] 이는한사본의소스에는변경이일어나지만, 더이상서로동기화되지않는다는것을의미합니다. (2) 원본설계가버그수정이나새로운기능으로개선되었을때사본을업데이트하는것이번거롭습니다.

더나은접근방식은그오픈소스회로를라이브러리로공개하는것입니다. 컴파일된 Chisel 코드는단순히 Java 클래스파일입니다. 그리고그클래스파일들은플랫폼에독립적입니다. 따라서이는 Chisel 라이브러리를공유하는이상적인방법입니다. Java(그리고 Scala) 는고유한그룹식별자와버전관리를통해라이브러리를공개적으로공유하는것을지원하는오랜전통과우수한인프라를갖추고있습니다. 이것이또한 Chisel 자체와일부지원라이브러리들이공개되는방식이기도합니다.

이절에서는 Chisel 라이브러리를공개하는데필요한단계들을설명합니다. 공개에사용하는도구들은빠르게변경될수있으므로, 인터넷에서최신정보를찾아보시기바랍니다. 이주제에관한좋은블로그게시물은여기에서찾을수있습니다.

[1] 저는이문구를 Java 의안전필수사양을개발하는토론세션에서 Doug Locke 로부터배웠습니다

16.1.1 라이브러리사용하기

Chisel 라이브러리는 build.sbt 에추가함으로써프로젝트에서사용할수있습니다. 다음은 ip-contributions에있는 Chisel 회로모음의예시입니다:

```
libraryDependencies += ”edu.berkeley.cs” % ”ip-contributions” % ”0.5.0”
```

ip-contributions 라이브러리에는이책에서설명한 UART 와 FIFO 도포함되어있습니다. 현대적인 IDE 는 build.sbt 에설정해두면검토를위해라이브러리의소스코드를자동으로다운로드할수있게해줍니다.

공유하고싶은 Chisel 회로가있다면 ip-contributions 에기여하는것을고려해보십시오. 기여는여러분의추가분에대한 git 풀리퀘스트로시작됩니다. 이는우호적인검토과정을시작하게됩니다.

16.1.2 사전준비사항

Maven Central은소프트웨어라이브러리를호스팅하는가장큰저장소중하나입니다. Maven Central 에게시하는가장쉬운방법은 Sonatype을통하는것입니다. Sonatype 은 Sonatype Repository를통해오픈소스프로젝트에대한무료호스팅을제공합니다. 라이브러리를게시하기전에다음과같은초기단계가필요합니다:

1. Sonatype JIRA 계정 만들기
2. 고유한 groupId 가필요한데, 이는보통도메인이름을역순으로나열한것입니다 (예: edu.berkeley.cs). GitHub 계정을 groupId 로사용할수도있는데, 예를들어제것은 io.github.schoeberl 입니다. 이 groupId 는 이슈를열어서등록합니다. 이는요청한 groupId 를 소유하고있음을증명해야하는수동절차입니다. GitHub 도메인이름을사용하는경우, 소유권을보여주기위해레지스터리를설정하도록요청받습니다.
3. Sonatype 로그인정보를담은 sonatype.sbt 를 $HOME/.sbt/1.0 에만드십시오:

```
credentials += Credentials(
    ”Sonatype Nexus Repository Manager”,
    ”oss.sonatype.org”,
    ”<user name>”,
    ”<password>”
)
```

4. 모든아티팩트는 PGP 키쌍으로서명되어야합니다. 오픈소스인 GNU Privacy Guard를사용할수있습니다. 다음명령으로공개 PGP 키를생성, 목록조회, 업로드할수있습니다:

```
gpg --gen-key
gpg --list-keys
gpg --keyserver keyserver.ubuntu.com --send-keys keyID
```

keyID 는키목록에서찾을수있는긴 16 진수문자열임에유의하십시오. 이단계가실패하면공개키를키서버에수동으로업로드할수있습니다.

16.1.3 라이브러리설정

각라이브러리마다다음을설정해야합니다:

1. project/plugins.sbt 에 sbt 플러그인설치하기:

```
addSbtPlugin("org.xerial.sbt" % "sbt-sonatype" % "2.3")
addSbtPlugin("com.jsuereth" % "sbt-pgp" % "2.0.2")
```

2. build.sbt 에라이브러리에대한정보를추가하십시오. 예로, ip-contributions 의 build.sbt에서해당부분은다음과같습니다:

```
name := "ip-contributions"

version := "0.4.0"

// groupId, SCM, license information
organization := "edu.berkeley.cs"
homepage := Some(url("https://github.com/freechipsproject/ip-contributions"))
scmInfo := Some(ScmInfo(url(
  "https://github.com/freechipsproject/ip-contributions"),
  "git@github.com/freechipsproject/ip-contributions"))
developers := List(Developer("schoeberl", "schoeberl",
  "martin@jopdesign.com", url("https://github.com/schoeberl")))
licenses += ("Unlicense", url("https://unlicense.org/"))
publishMavenStyle := true

// disable publish with Scala version
crossPaths := false
```

```
publishTo := Some(
  if (isSnapshot.value)
    Opts.resolver.sonatypeSnapshots
  else
    Opts.resolver.sonatypeStaging
)
```

16.1.4 정기게시

이제다음명령으로라이브러리에서명하고 Sonatype 에게시할준비가모두되었습니다:

```
sbt publishSigned
```

제설정에서는 sbt 의서명이 (가끔) 작동하지않아서, 서명을위해 pgp 명령을따로복사해서실행해야하는데, 대략다음과비슷합니다:

```
gpg --detach-sign --armor --use-agent --output path-to.asc\\
path-to-0.1.pom
```

그리고 sbt publishSigned 를반복하십시오.

이미예를들어내부프로젝트에서 Sonatype 으로부터그라이브러리를사용할수있습니다. 그러나라이브러리를 Maven Central 에릴리스하려면다음을실행하십시오:

```
sbt sonatypeRelease
```

몇분후면 Maven Central Repository Search에서확인할수있을것입니다.

16.2 Chisel 에기여하기

다음은고급주제이며, 첫번째풀리퀘스트 (PR) 를만들기전에먼저 GitHub 의 Chisel 프로젝트이슈에대한논의를따라가볼것을제안합니다.

16.2.1 개발환경설정하기

Chisel 은여러개의서로다른저장소로구성됩니다. 주저장소들은 CHIPS Alliance에서호스팅되고, 그외에는 GitHub 의 freechips 조직에서호스팅됩니다.

기여하고자하는저장소를여러분의개인 GitHub 계정으로포크하십시오. GitHub 웹인터페이스에서 Fork 버튼을눌러저장소를포크할수있습니다. 그런다음해당포크로부터저장소의포크를클론하십시오.[2] 여기예제에서는 chisel3 를변경하며, 제로컬포크에대한클론명령은다음과같습니다:

```
$ git clone git@github.com:schoeberl/chisel3.git
```

Chisel 3 를컴파일하고로컬라이브러리로게시하려면다음을실행하십시오:

```
$ cd chisel3
$ sbt compile
$ sbt publishLocal
```

publish local 명령을실행할때게시된라이브러리의버전문자열을주의깊게살펴보십시오. 이버전문자열에는 SNAPSHOT 이라는문자열이포함되어있습니다. 테스터를사용하는데게시된버전이 Chisel SNAPSHOT 과호환되지않는다면, chisel-tester 저장소도포크하고클론하여로컬에게시하십시오.

Chisel 에서변경사항을테스트하려면아마도 Chisel 프로젝트도설정해야할것입니다. 예를들어 빈 Chisel 프로젝트를포크/클론한뒤이름을변경하고그로부터 .git 폴더를제거하는방식으로설정할수있습니다.

build.sbt 를변경하여로컬에게시된버전의 Chisel 을참조하도록하십시오. Chisel 테스트애플리케이션을컴파일하고, 이것이로컬에게시된 Chisel 라이브러리버전을사용하는지자세히확인하십시오 (SNAPSHOT 버전도게시되어있으므로, 예를들어여러분의 Chisel 라이브러리와애플리케이션코드간에 Scala 버전이다를경우, 로컬에게시된라이브러리대신서버의 SNAPSHOT 버전을사용하게됩니다).

Chisel 저장소의몇가지참고사항도참조하십시오.

16.2.2 테스트

Chisel 라이브러리를변경할때는 Chisel 테스트를실행해야합니다. sbt 기반프로젝트에서는보통다음과같이실행됩니다:

[2] FIRRTL/Chisel 의호환성이깨지는변경일경우, firrtl 도포크하고클론해야할수있음에유의하십시오.

```
$ sbt test
```

또한 Chisel 에기능을추가하는경우, 새기능에대한테스트도제공해야합니다.

16.2.3 Pull Request 로기여하기

Chisel 프로젝트에서는어떤개발자도주저장소에직접커밋하지않습니다. 기여는라이브러리의포크된버전의브랜치로부터 풀리퀘스트를통해조직됩니다. 더자세한정보는 GitHub 의 풀리퀘스트를통한협업에관한문서를참조하십시오. Chisel 그룹은 기여가이드라인을문서화하기시작했습니다.

16.3 연습문제

UInt 타입을위한새로운연산자를고안하고, 이를 Chisel 라이브러리에구현한다음, 그연산자를탐구할수있는사용/테스트코드를작성하십시오. 유용한연산자일필요는없으며, 아무것이나좋습니다. 예를들어좌변이 0 이아니면좌변을반환하고, 그렇지않으면우변을반환하는? 연산자같은것이면됩니다. 멀티플렉서처럼들리지않습니까? 이를추가하는데몇줄의코드가필요했습니까?[3]

이것이아무리간단해보이더라도, Chisel 프로젝트를포크하여여러분의작은확장을추가하고싶은유혹에빠지지마십시오. 변경과확장은주개발자들과조율되어야합니다. 이연습문제는그저여러분을시작하게하기위한간단한연습에불과합니다.

대담해지고싶다면, 공개이슈 중하나를골라해결을시도해볼수있습니다. 그런다음풀리퀘스트를통해 Chisel 에기여하십시오. 다만, 아마도먼저 GitHub 저장소들을관찰하며 Chisel 의개발스타일을살펴보는것이좋을것입니다. Chisel 오픈소스프로젝트에서변경사항과풀리퀘스트가어떻게처리되는지살펴보십시오.

[3] 빠르고간단한구현에는단두줄의 Scala 코드만필요함

17 요약

이책은하드웨어구성언어 Chisel 을이용한디지털설계입문을제시했습니다. 우리는 Chisel 로기술된여러개의간단한규모부터중간규모의디지털회로를살펴보았습니다. Chisel 은 Scala 안에내장되어있으며, 따라서 Scala 의강력한추상화를물려받습니다. 이책은입문서로의도되었으므로, Scala 의간단한사용예제로범위를제한했습니다. 다음으로논리적인단계는 Scala 의몇가지기초를배우고이를여러분의 Chisel 프로젝트에적용하는것입니다.

이책에대한피드백을받는다면기쁠것이며, 이를바탕으로책을더개선하고새로운판을출간하겠습니다. mailto:masca@dtu.dk로저에게연락하시거나 GitHub 저장소에이슈를등록해주십시오. 또한이책의저장소에대한수정및개선을위한풀리퀘스트도기꺼이받습니다.

소스접근

이책은오픈소스로제공되며, 여기에는제시된모든 Chisel 코드가포함됩니다. 이저장소에는또한 Chisel 을이용한디지털설계강좌용슬라이드와모든 Chisel 예제가포함되어있습니다: https://github.com/schoeberl/chisel-book

이책에서참조되는중간규모예제모음또한오픈소스로제공됩니다. 이모음에는다양한인기 FPGA 보드용프로젝트도포함되어있습니다: https://github.com/schoeberl/chisel-examples

A VHDL 과 Verilog

이책은 Chisel 을이용한디지털설계를가르칩니다. 그러나여러분의전문적인경력에서다른하드웨어기술언어를접하게될수도있습니다. 디지털설계의기본개념과 Chisel 로디지털회로를기술하는방법을이해하고나면, 다른언어로전환하는데는며칠밖에걸리지않을것입니다.

지배적인두가지하드웨어기술언어는 VHDL 과 Verilog 이며, SystemVerilog(SV) 로업데이트되었습니다. 하드웨어개발및테스트에서최근의움직임은 VHDL 에서 SV 로이동하는것입니다. 그러나많은오픈소스도구가여전히표준 Verilog 를주로지원하므로, 예제에서는평범한 Verilog 를사용하겠습니다.

Chisel, VHDL, Verilog 로작성된코드예제를보여드리겠습니다. 이부록은 VHDL 이나 Verilog 에대한완전한소개가아님을유의하십시오. 이는여러분이 VHDL 과 Verilog 를읽기시작할수있도록하고, Chisel 에서 VHDL 과 Verilog 로하드웨어구성체를번역할수있도록해줄뿐입니다. Chisel 에서 Verilog 로의번역을위해서는 Chisel 이생성한 Verilog 에서시작할수도있습니다. 손으로작성한코드보다가독성이떨어질수있지만, 동작하는출발점이됩니다. VHDL 과 SystemVerilog 를나란히잘소개한자료는 [10] 에서찾을수있습니다. 나아가, Verilog 로된레거시코드를 Chisel 설계에통합하는방법도보여드리겠습니다.

A.1 코드예제

VHDL 과 Verilog 예제를 Chisel 버전의코드와함께제시하겠습니다.

A.1.1 컴포넌트

목록 A.1는 Chisel 로작성된예제컴포넌트로서간단한가산기[1]를보여줍니다. 완전한예제로만들기위해, 목록에는 import 문도포함되어있습니다.

[1] 함수를정의하는코드가단몇줄, 이예제에서는단한줄에불과할때는실제설계에서이렇게작은컴포넌트를절대사용해서는안됩니다. 그러나컴포넌트라는개념을소개하기위해간결하게유지합니다.

```
import chisel3._
import chisel3.util._

class ChiselAdder extends Module {
  val io = IO(new Bundle() {
    val a, b = Input(UInt(8.W))
    val sum = Output(UInt(8.W))
  })
  io.sum := io.a + io.b
}
```

Listing A.1: Chisel 로작성된간단한컴포넌트.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity adder is
  port (
    a, b : in unsigned(7 downto 0);
    sum : out unsigned(7 downto 0)
  );
end entity;



architecture rtl of adder is
begin
  sum <= a + b;
end architecture;
```

Listing A.2: VHDL 로작성된간단한컴포넌트.

```
module adder(
  input  [7:0] a,
  input  [7:0] b,
  output [7:0] sum
);

  assign sum = a + b;

endmodule
```

Listing A.3: Verilog 로작성된간단한컴포넌트.

목록 A.2는 VHDL 로작성된가산기컴포넌트를보여줍니다. Chisel 과마찬가지로, 파일은몇가지표준함수를포함하기위한 import 문으로시작합니다. VHDL 에서컴포넌트는 entity 선언과 architecture 정의가필요합니다. entity 는컴포넌트의포트를선언합니다. architecture 는컴포넌트의기능을정의합니다. architecture 는이름이지정되어야하는데, 예제에서그이름은 rtl 입니다. 그러나오늘날그러한명명은실질적인의미를잃었습니다. VHDL 은대소문자를구분하지않으므로, Adder 와 adder 는동일하게취급됩니다. 그러나현재의관행은식별자에소문자를고수하는것입니다.

목록 A.3는 Verilog 로작성된가산기컴포넌트를보여줍니다. Chisel 과마찬가지로, 컴포넌트는모듈이라고도불립니다.[2] 가산기는두개의 8 비트입력과하나의 8 비트출력을가집니다. assign 문은 a 와 b 의합의값을출력포트 sum 에지속적으로할당하는동시적문입니다. 이러한단순한컴포넌트에서는 Verilog 와 Chisel 의차이가미미합니다.

A.1.2 컴포넌트사용하기

목록 A.4는 Chisel 에서 new 로컴포넌트를생성하고, 이름 (m) 을부여한뒤, 입출력포트를와이어에연결하는방법을보여줍니다.

목록 A.5는 VHDL 에서 adder 컴포넌트를선언하고사용하는방법을보여줍니다. VHDL 은강타입언어이므로, 컴포넌트는인스턴스화되기전에선언되어야합니다. 컴포넌트선언은가산기자체의 entity 선언과매우유사합니다. 컴포넌트는이를호출하고, 이름 (m) 을앞에붙이며, port map 으로입출력포트를신호에연결함으로써인스턴스화됩니다.

[2] 사실 Chisel 의명명은 Verilog 에서영감을받았습니다.

```
  val in1 = Wire(UInt(8.W))
  val in2 = Wire(UInt(8.W))
  val result = Wire(UInt(8.W))

  val m = Module(new ChiselAdder)
  m.io.a := in1
  m.io.b := in2
  result := m.io.sum
```

Listing A.4: Chisel 에서 ChiselAdder 컴포넌트사용하기.

```
  signal in1, in2, result : unsigned(7 downto 0);

  component adder
    port (
      a, b : in unsigned(7 downto 0);
      sum : out unsigned(7 downto 0)
    );
  end component;

begin

    m: adder
        port map (
        a => in1,
        b => in2,
        sum => result
        );
```

Listing A.5: VHDL 에서 adder 컴포넌트사용하기.

```
    wire [7:0] in1;
    wire [7:0] in2;
    wire [7:0] result;

    adder m(.a(in1), .b(in2), .sum(result));
```

Listing A.6: Verilog 에서 adder 컴포넌트사용하기.

```
val reg = RegEnable(data, 0.U(8.W), enable)
```

Listing A.7: Chisel 에서리셋과인에이블을가진레지스터.

```
    signal reg : std_logic_vector(7 downto 0);

begin
    process (clock)
    begin
        if rising_edge(clock) then
            if reset = '1' then
                reg <= (others => '0');
            elsif enable = '1' then
                reg <= data;
            end if;
        end if;
    end process;
```

Listing A.8: VHDL 에서리셋과인에이블을가진레지스터.

목록 A.6는 Verilog 에서컴포넌트 (모듈) 를생성하고사용하는방법을보여줍니다. 여기서는이를호출하고, 이름 (m) 을뒤에붙인뒤, 와이어를포트에연결함으로써생성됩니다.

a, b, sum 은가산기의포트를나타내며, in1, in2, result 는그포트에연결된와이어/신호임을유의하십시오. 세예제모두에서가산기의인스턴스는 m 이라는이름이붙었습니다. VHDL 과 Verilog 에서는컴포넌트의이름이그다지자주사용되지않지만, Chisel 에서는 IO 포트에접근하기위해모듈의이름이필요합니다.

A.1.3 레지스터

목록 A.7는 RegEnable 의세매개변수버전을사용하여, Chisel 에서리셋 (0 으로) 과인에이블입력을가진 8 비트레지스터를보여줍니다. 입력은생성시점에연결되며, 출력은 RegEnable 의반환값으로사용할수있습니다. clock 과 reset 신호는 Chisel 에서암묵적임을유의하십시오.

목록 A.8는동기리셋과인에이블입력을가진레지스터 (reg) 에대한 VHDL 코드를보여줍니다. VHDL 의 process 는민감도목록에 clock 신호를포함함을유의

```
reg [7:0] reg_data;

always @(posedge clk) begin
  if (reset)
    reg_data <= 8'b0;
  else if (enable)
    reg_data <= data;
end
```

Listing A.9: Verilog 에서리셋과인에이블을갖춘레지스터.

하십시오. 레지스터는클록된프로세스와 rising_edge(clock) 을사용하여추론됩니다. Chisel 과달리, clock 과 reset 둘다명시적으로사용되어야합니다.

Listing A.9은 Verilog 에서동기식리셋과인에이블입력을갖춘레지스터 (reg_data) 의정의를보여줍니다. Verilog 의 reg 키워드는변수를선언합니다. 이것이반드시그변수가레지스터를나타낸다는것을의미하지는않습니다. always @(*) 블록에서사용될때는조합논리를나타낼수도있습니다. 클록을사용하는 always @(posedge clk) 문은해당블록내부의코드가레지스터를나타낸다는것을암시합니다.

Verilog 와 VHDL 에서는레지스터가코드패턴에의해암묵적으로정의됩니다. Chisel 에서는레지스터가명시적으로정의됩니다.

A.1.4 조합블록

세언어모두에서단순한조합논리는동시문 (concurrent statement) 으로서의 할당을통해작성할수있습니다. 가산기컴포넌트에서그예를이미살펴본바있습니다. 그러나중첩된조건을기술하는것과같이더복잡한논리의경우에는조합논리를 위한블록을사용하는것이더편리합니다. Verilog 에서는이것이 always 블록이고, VHDL 에서는 process 입니다.

Listing A.10은조합블록의예시로 Chisel 의 switch 문을보여줍니다. 이코드는 4:1 멀티플렉서입니다. Chisel 에서는출력의기본값이 switch 문이전에할당되어야합니다.

Listing A.11은 VHDL 에서동일한조합블록을보여줍니다. 조합블록은 process 입니다. 시작은추가로 begin 으로표시되고프로세스의끝은 end process; 로표시됩니다. 해당 case 문의기본값은 when others 항목에할당됩니다. VHDL 프로세스에는감지목록 (sensitivity list) 이있으며, 이목록에는표현식이나조

```
io.out := 0.U
switch(io.sel) {
  is("b00".U) { io.out := io.in(0) }
  is("b01".U) { io.out := io.in(1) }
  is("b10".U) { io.out := io.in(2) }
  is("b11".U) { io.out := io.in(3) }
}
```

Listing A.10: Chisel 의 switch 문.

```
process (sel, input)
begin
  case sel is
    when "00" => output <= input(0);
    when "01" => output <= input(1);
    when "10" => output <= input(2);
    when "11" => output <= input(3);
    when others => output <= '0';
  end case;
end process;
```

Listing A.11: VHDL 의 case 문.

```
module comb(
  input  [1:0] sel,
  input  [3:0] in,
  output reg out
);

  always @(*) begin
    case (sel)
      2'b00: out = in[0];
      2'b01: out = in[1];
      2'b10: out = in[2];
      2'b11: out = in[3];
      default: out = 1'b0;
    endcase
  end

endmodule
```

Listing A.12: Verilog 의 case 문.

건의우변에나타나는모든신호가나열되어야함에유의하십시오. 이예제에서는 sel 과 input 이이에해당합니다.

Listing A.12은 Verilog 에서동일한조합블록을보여줍니다. 조합블록의시작이 always @(*) begin 으로표시되고블록의끝이 end 로표시되는점에유의하십시오. 해당 case 문의기본값은 default 항목에할당됩니다. 이예제에는모듈헤더가포함되어있어, 출력 out 이 always 블록에서할당되므로 reg 로선언되어야함을보여줍니다.

Verilog 에는두가지서로다른할당연산자가있음에유의하십시오: = 는 블로킹 *(blocking)* 문이라불리고, <= 는 논블로킹 *(non-blocking)* 문이라불립니다. 이는혼동의원인이될수도있습니다. 레지스터 (always @(posedge clk) 블록내) 에는 <= 할당연산자를사용하고, 조합논리 (always @(*) 블록내) 에는 = 할당연산자를사용하십시오.

조합블록에서또다른편리한구성요소는조건문의조합으로, "if...else if...else" 문이라고도알려져있습니다. 세개의입력 (in1, in2, in3), 두개의불리언조건 (c1 과 2), 그리고출력 out 을사용하는예제를보여드리겠습니다.

Listing A.13는 Chisel 버전을보여줍니다. 그러나 Scala 에서는 if 와 else 가조건식 (예약키워드) 이므로, Chisel 은 when, .elsewhen, .otherwise 를사용

```
when (io.c1) {
  io.out := io.in1
} .elsewhen (io.c2) {
  io.out := io.in2
} .otherwise {
  io.out := io.in3
}
```

Listing A.13: Chisel 의"if...else if...else" 문.

```
process(c1, c2, in1, in2, in3)
begin
  if c1 = '1' then
    output <= in1;
  elsif c2 = '1' then
    output <= in2;
  else
    output <= in3;
  end if;
end process;
```

Listing A.14: VHDL 의"if...else if...else" 문.

합니다.

Listing A.14는 VHDL 에서동일한구성을보여줍니다. (조건신호를포함한) 모든입력신호가프로세스의감지목록에포함되어야함에유의하십시오.

Listing A.15는 always 블록내에서 Verilog 로작성된동일한표현을보여줍니다.

A.1.5 고급 Chisel 기능

Chisel, Verilog, VHDL 에서기본요소들은비슷해보이며, 장황함의정도도비슷한범위에있습니다 (다만 VHDL 이약간더수다스럽습니다). 그러나 VHDL 과 Verilog 어느쪽도하드웨어기술을위한객체지향기능을포함하고있지않습니다. SystemVerilog 는테스트벤치작성을위해서만객체지향프로그래밍을포함하고있습니다. 두언어모두재사용가능한하드웨어생성기를작성할때중요한함수

```
always @(*) begin
  if (c1)
    out = in1;
  else if (c2)
    out = in2;
  else
    out = in3;
end
```

Listing A.15: Verilog 의"if...else if...else" 문.

형프로그래밍이빠져있습니다. 게다가방향을가지는 Bundle(번들) 과방향을뒤집을수있는가능성또한빠져있습니다.

어떤하드웨어든 Chisel, VHDL, Verilog 에서동등하게기술될수있지만, Chisel 의현대적인기능들은하드웨어생성기를작성함으로써더높은수준의추상화로프로그래밍하는것을가능하게합니다.

A.2 외부모듈과레거시코드의통합

TODO: Verilog 와 VHDL 을 (GHDL 을사용하여) 기술하기

때로는레거시코드처럼 Verilog 로작성된설명을가진컴포넌트를포함하고싶을수도있고, 또는컴포넌트에서생성되는 Verilog 가여러분의합성도구가인식하여사용가능한프리미티브에매핑할수있는구체적인구조를갖도록보장하고싶을수도있습니다. Chisel 은 BlackBox 와 ExtModule 클래스를통해이를지원하며, 이를통해 Verilog 소스로컴포넌트를정의할수있습니다. 둘다 Map[String, Param] 으로매개변수화되며, 이는생성되는 Verilog 에서모듈매개변수로변환됩니다. BlackBox 는개별 Verilog 파일로생성되는반면, ExtModule 은플레이스홀더역할을하며소스가없는모듈인스턴스화로생성됩니다. 이기능덕분에 ExtModule 은예를들어클록버퍼나입력버퍼와같은 Xilinx 나 Intel 디바이스프리미티브에특히유용합니다.

```
class BUFGCE extends BlackBox(Map("SIM_DEVICE" ->
    "7SERIES")) {
  val io = IO(new Bundle {
    val I = Input(Clock())
    val CE = Input(Bool())
```

```
    val O = Output(Clock())
  })
}


class alt_inbuf extends ExtModule(
  Map("io_standard" -> "1.0 V",
    "location" -> "IOBANK_1",
    "enable_bus_hold" -> "on",
    "weak_pull_up_resistor" -> "off",
    "termination" -> "parallel 50 ohms")) {
  val io = IO(new Bundle {
    val i = Input(Bool())
    val o = Output(Bool())
  })
}
```

반면에 BlackBox 는어떤컴포넌트든표현할수있습니다. 이는세가지다른방식으로선언될수있으며, 소스는인라인으로포함되거나별도의파일로제공될수있습니다. 예시로, 다음과같은 IO 를가진 32 비트가산기를살펴보겠습니다.

```
class BlackBoxAdderIO extends Bundle {
  val a = Input(UInt(32.W))
  val b = Input(UInt(32.W))
  val cin = Input(Bool())
  val c = Output(UInt(32.W))
  val cout = Output(Bool())
}
```

인라인버전은다음과같이선언됩니다:

```
class InlineBlackBoxAdder extends HasBlackBoxInline {
  val io = IO(new BlackBoxAdderIO)
  setInline("InlineBlackBoxAdder.v",
    s"""
       |module InlineBlackBoxAdder(a, b, cin, c, cout);
       |input  [31:0] a, b;
       |input  cin;
       |output [31:0] c;
       |output cout;
       |wire   [32:0] sum;
       |
```

```
      |assign sum  = a + b + {31'b0, cin};
      |assign c    = sum[31:0];
      |assign cout = sum[32];
      |
      |endmodule
  """.stripMargin)
}
```

문자열리터럴내에 (큰따옴표앞에 s 나 f 로표시) 소스코드를제공하고파이프를사용하면깔끔하게서식이지정된 Verilog 코드를포함할수있습니다. 또한, Scala 변수를 $ 나 ${} 이스케이프문자를사용하여삽입할수있으므로매개변수화지원도가능해집니다. stripMargin 메서드는코드를생성할때파이프와탭을제거합니다.

인라인 BlackBox 에대한두가지대안이있으며, 둘다별도의파일에 Verilog 소스가있을것으로예상합니다. 이들은다음과같이선언됩니다.

```
class ResourceBlackBoxAdder extends HasBlackBoxResource {
  val io = IO(new BlackBoxAdderIO)
  addResource("/ResourceBlackBoxAdder.v")
}


class PathBlackBoxAdder extends HasBlackBoxPath {
  val io = IO(new BlackBoxAdderIO)
  addPath("./src/main/resources/PathBlackBoxAdder.v")
}
```

HasBlackBoxResource 버전은 ./src/main/resource 폴더에서 Verilog 소스를찾을것으로예상합니다. HasBlackBoxPath 버전은프로젝트폴더로부터의임의의상대경로가제공될수있습니다.

BlackBox 는 Module(new BlackBoxModule) 로감싸서다른모듈과마찬가지로인스턴스화됩니다. 이는직접테스트할수없으며, 테스터가이름이지정된클래스나익명클래스로감싸야합니다. 둘다 Blackbox 와동일한 IO 를가질수있습니다.

```
  class InlineAdder extends Module {
    val io = IO(new BlackBoxAdderIO)
    val adder = Module(new InlineBlackBoxAdder)
    io <> adder.io
  }
```

```
test(new Module {
  val io = IO(new BlackBoxAdderIO)
  val adder = Module(new InlineBlackBoxAdder)
  io <> adder.io
})
```

HasBlackBoxInline, HasBlackBoxPath, HasBlackBoxResource 는 Chisel 의 BlackBox 클래스를확장하는 trait 임에유의하십시오. 즉, 예를들어 class Example extends BlackBox with HasBlackBoxInline 은
class Example extends HasBlackBoxInline 과동등합니다.

A.3 연습문제

여러분의 Chisel 설계중하나를사용하여수동으로 Verilog 로변환해보십시오. Verilog 로테스트벤치를작성하지않으려면, ChiselTest 에서작성한테스트코드를재사용하고 Verilog 코드를 black box 로인스턴스화하여테스트할수있습니다. 또한테스트코드를변경하여 Chisel 컴포넌트와 Verilog 컴포넌트를모두인스턴스화하고동일한 Chisel 테스터에서비교할수도있습니다. 이책의예제들은그런방식으로테스트됩니다. 이를어떻게수행할수있는지알아보려면책예제의코드를살펴보십시오. 이러한형태의코시뮬레이션은무작위테스트벡터로구동하여 Chisel ” 골든모델” 과여러분의 Verilog 변환결과를비교하는방식으로도수행할수있습니다.

Chisel/Verilator 내에서의 VHDL 통합은아직그리매끄럽지않지만, yosys 용 GHLD 플러그인을사용하는방식이 VHDL 을 Verilog 로변환할수있어야하며, 이후 Verilog 설계와마찬가지로 Chisel 에서테스트할수있습니다.

VHDL 의경우사용가능한오픈소스옵션이더적습니다. 이장의코드는 (copilot 이생성한) VHDL 로작성된테스트벤치로테스트되었으며 GHDL 로시뮬레이션되었습니다.[3]

[3] http://ghdl.free.fr/

B 예약된키워드

Chisel(및 Scala) 에는여러예약된키워드가있으며, 이는여러분의하드웨어설계에서식별자로사용할수없습니다. 표 B.1는 Scala 의예약어를나열합니다.

:	<-	<:	<%	=	=>
>:	@	#	_	abstract	case
catch	class	def	do	else	extends
false	final	finally	for	forSome	if
implicit	import	lazy	match	new	null
object	override	package	private	protected	return
sealed	super	this	throw	trait	true
try	type	val	var	while	with
yield					

Table B.1: Scala 언어의예약된키워드.

표 B.2는 Chisel 라이브러리가추가한예약어를나열합니다. Scala 예약어목록과달리, 여기에는 Chisel 이정의한타입/클래스이름도포함됩니다. 기술적으로는가능하지만, 예를들어 + 나 « 와같은 Chisel(및 Scala) 연산자도사용을피해야합니다.

:=	##	andR	Bits	Bool	Cat
clock	elsewhen	io	is	Mem	Module
name	orR	otherwise	Reg	RegEnable	RegInit
RegNext	reset (리셋)	SInt	switch	SyncMem	UInt
Vec	VecInit	when	Wire	WireDefault	xorR

Table B.2: Chisel 언어의예약된키워드.

C Chisel 프로젝트

Chisel 은아직많은프로젝트에서사용되지않고있습니다. 따라서이언어와코딩스타일을익히기위한오픈소스 Chisel 코드는드뭅니다. 여기서는우리가알고있는, Chisel 을사용하는오픈소스프로젝트몇가지를나열합니다.

Rocket Chip Rocket 마이크로아키텍처와 TileLink 인터커넥트생성기로구성된 RISC-V [31] 프로세서컴플렉스생성기입니다. 원래 UC 버클리에서최초의칩규모 Chisel 프로젝트로개발되었으며 [4], Rocket Chip 은현재 SiFive에의해상업적으로지원되고있습니다.

Sodor 교육용으로만들어진 RISC-V 구현모음입니다. 1 단계, 2 단계, 3 단계, 5 단계파이프라인구현을포함합니다. 모든프로세서는명령어페치, 데이터접근, 디버그포트를통한프로그램로딩이공유하는단순한스크래치패드메모리를사용합니다. Sodor 는주로시뮬레이션에서사용하도록의도되었습니다.

Patmos 실시간시스템에최적화된프로세서의구현입니다 [27]. Patmos 저장소에는시간예측가능한메모리아비터 [23], 네트워크온칩 [26], 소유권을가진공유스크래치패드메모리 [28] 등여러멀티코어통신아키텍처가포함되어있습니다.

FlexPRET 정밀타이밍아키텍처의구현입니다 [34]. FlexPRET 는 RISC-V 명령어집합을구현하며 Chisel 3.1 로업데이트되었습니다.

Lipsi 시스템온칩에서의유틸리티기능을위한소형프로세서입니다 [19]. Lipsi 의코드베이스는매우작기때문에, Chisel 에서프로세서설계를시작하는쉬운출발점으로활용될수있습니다. Lipsi 는또한 Chisel/Scala 의생산성을보여줍니다. 필자는 Chisel 로하드웨어를기술하여 FPGA 에서구동하고, Scala 로어셈블러를작성하고, 공동시뮬레이션을위해 Scala 로 Lipsi 명령어집합시뮬레이터를작성하고, Lipsi 어셈블러로몇가지테스트케이스를작성하는데 14 시간이걸렸습니다.

Leros 소형어큐뮬레이터기반프로세서로, 원래는 VHDL 로작성된 16 비트버전이었습니다 [18]. 재설계된버전은이제 Chisel 로작성되었으며 32

비트버전입니다 [25]. Morten Borup Petersen 은 Leros ISA 를지원하도록 LLVM C 컴파일러를이식했습니다 [15].

OpenSoC Fabric Chisel 로작성된오픈소스 NoC 생성기입니다 [9]. 대규모설계탐색을위한시스템온칩을제공하기위해만들어졌습니다. 이 NoC 는웜홀라우팅, 흐름제어를위한크레딧, 가상채널을갖춘최신설계입니다. OpenSoC Fabric 은여전히 Chisel 2 를사용하고있습니다.

DANA Rocket Custom Coprocessor(RoCC) 인터페이스를사용하여 RISC-V Rocket 프로세서와통합되는신경망가속기입니다 [7] [8]. DANA 는추론과학습을모두지원합니다.

Chiselwatt POWER Open ISA 의구현입니다. Micropython 을구동하기위한명령어를포함하고있습니다.

VTA Hardware Design Stack Apache TVM 머신러닝컴파일러프레임워크를위한머신러닝가속기입니다.

Chisel IP Contributions 소형 Chisel 컴포넌트들을모으기시작한프로젝트입니다. 이책에서소개한 UART 와 FIFO 의소스코드를포함하고있습니다.

Constellation UC Berkeley 에서개발된 NoC 생성기입니다 [33]. Constellation 은웜홀라우팅, 가상채널, 크레딧기반흐름제어를갖춘패킷교환 NoC 를생성합니다. NoC 로의인터페이스는 AXI-4 또는 TileLink 를사용할수있습니다. 다른 NoC 및 NoC 생성기와비교했을때, Constellation 은애플리케이션특화경로를가진어떠한토폴로지도생성할수있습니다.

XiangShan 오픈소스비순차 (out-of-order) RISC-V 프로세서입니다 [32]. XiangShan 은애자일하드웨어설계를위해 Chisel 을장려하고있습니다.

Wildcat 단순한파이프라인 RISC-V 프로세서입니다 [20]. Wildcat 의초점은교육에활용할수있는읽기쉬운 Chisel 코드를제공하는데있습니다.

Chisel 을사용하는오픈소스프로젝트를알고계시다면, 이책의다음판에포함할수있도록필자에게알려주시기바랍니다.

D 약어

하드웨어설계자와컴퓨터엔지니어는약어를즐겨사용합니다. 하지만익숙해지는데는시간이걸립니다. 다음은흔히쓰이는디지털설계및컴퓨터아키텍처용어목록입니다.

ADC 아날로그-디지털변환기

ALU 산술논리장치

ASIC 주문형반도체

CFG 제어흐름그래프

Chisel Scala 내장언어로하드웨어를구성하기

CISC 복합명령어집합컴퓨터

CPI 명령어당클록사이클수

CPU 중앙처리장치

CRC 순환중복검사

DAC 디지털-아날로그변환기

DFF D 플립플롭, 데이터플립플롭

DMA 직접메모리접근

DRAM 동적임의접근메모리

EMC 전자기적합성

ESD 정전기방전

FF 플립플롭

FIFO 선입선출

FPGA 필드프로그래머블게이트어레이

HDL 하드웨어기술언어

HLS 고수준합성

IC 명령어수

IDE 통합개발환경

ILP 명령어수준병렬성

IC 집적회로

IO 입출력

ISA 명령어집합구조

JDK 자바개발키트

JIT 즉시컴파일

JVM 자바가상머신

LC 로직셀

LRU 최소최근사용

LSB 최하위비트

MMIO 메모리매핑 IO

MSB 최상위비트

MUX 멀티플렉서

OO 객체지향

OOO 비순차적

OS 운영체제

RAM 랜덤액세스메모리

RISC 축소명령어집합컴퓨터

SDRAM 동기식 DRAM

SRAM 정적랜덤액세스메모리

TOS 스택최상단

UART 범용비동기송수신기

VHDL VHSIC 하드웨어기술언어

VHSIC 초고속집적회로

Bibliography

[1] Altera. Avalon interface specification, April 2005.

[2] ARM. AMBA specification (rev 2.0), May 1999.

[3] ARM. AMBA AXI and ACE protocol specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite. https://developer.arm.com/documentation/ihi0022/e/, 2011.

[4] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a Scala embedded language. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference (DAC 2012)*, pages 1216–1225, San Francisco, CA, USA, June 2012. ACM.

[6] William J. Dally, R. Curtis Harting, and Tor M. Aamodt. *Digital design using VHDL: A systems approach*. Cambridge University Press, 2016.

[7] Schuyler Eldridge, Amos Waterland, Margo Seltzer, Jonathan Appavoo, and Ajay Joshi. Towards general-purpose neural

network computing. In *2015 International Conference on Parallel Architecture and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015*, pages 99–112, 2015.

[8] Schuyler Eldridge, Amos Waterland, Margo Seltzer, and Jonathan Appavooand Ajay Joshi. Towards general-purpose neural network computing. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 99–112, Oct 2015.

[9] Farzaf Fatollahi-Fard, David Donofrio, George Michelogiannakis, and John Shalf. Opensoc fabric: On-chip network generator. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 194–203, April 2016.

[10] S. Harris and D. Harris. *Digital Design and Computer Architecture, RISC-V Edition*. Elsevier Science, 2021.

[11] IBM. On-chip peripheral bus architecture specifications v2.1, April 2001.

[12] Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. Open-source formal verification for chisel. In *Proceedings of the Fourth Workshop on Open-Source EDA Technology (WOSET)*, 2021.

[13] OCP-IP Association. Open core protocol specification 2.1. http://www.ocpip.org/, 2005.

[14] David A. Patterson and John L. Hennessy. *Computer Organization and Design, RISC-V Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2020.

[15] Morten Borup Petersen. A compiler backend and toolchain for the leros architecture. B.sc.eng. thesis, Technical University of Denmark, 2019.

[16] Wade D. Peterson. WISHBONE system-on-chip (SoC) interconnection architecture for portable IP cores, revision: B.3. Available at http://www.opencores.org, September 2002.

[17] Martin Schoeberl. SimpCon - a simple and efficient SoC interconnect. In *Proceedings of the 15th Austrian Workshop on Microelectronics, Austrochip 2007*, Graz, Austria, October 2007.

[18] Martin Schoeberl. Leros: A tiny microcontroller for FPGAs. In *Proceedings of the 21st International Conference on Field Programmable Logic and Applications (FPL 2011)*, pages 10–14, Chania, Crete, Greece, September 2011. IEEE Computer Society.

[19] Martin Schoeberl. Lipsi: Probably the smallest processor in the world. In *Architecture of Computing Systems – ARCS 2018*, pages 18–30. Springer International Publishing, 2018.

[20] Martin Schoeberl. The educational risc-v microprocessor wildcat. In *Proceedings of the Sixth Workshop on Open-Source EDA Technology (WOSET)*, 2024.

[21] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.

[22] Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos reference handbook. Technical report, Technical University of Denmark, 2014.

[23] Martin Schoeberl, David VH Chong, Wolfgang Puffitsch, and Jens Sparsø. A time-predictable memory network-on-chip. In *Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014)*, pages 53–62, Madrid, Spain, July 2014.

[24] Martin Schoeberl, Hans Jakob Damsgaard, Luca Pezzarossa, Oliver Keszocze, and Erling Rennemo Jellum. Hardware generators with chisel. In *2024 27th Euromicro Conference on Digital System Design (DSD)*, pages 168–175, 2024.

[25] Martin Schoeberl and Morten Borup Petersen. Leros: The return of the accumulator machine. In Martin Schoeberl, Thilo Pionteck, Sascha Uhrig, Jürgen Brehm, and Christian Hochberger, editors, *Architecture of Computing Systems - ARCS 2019 - 32nd International Conference, Proceedings*, pages 115–127. Springer, 1 2019.

[26] Martin Schoeberl, Luca Pezzarossa, and Jens Sparsø. A minimal network interface for a simple network-on-chip. In Martin Schoeberl, Thilo Pionteck, Sascha Uhrig, Jürgen Brehm, and Christian Hochberger, editors, *Architecture of Computing Systems - ARCS 2019*, pages 295–307. Springer, 1 2019.

[27] Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. Patmos: A time-predictable microprocessor. *Real-Time Systems*, 54(2):389–423, Apr 2018.

[28] Martin Schoeberl, Tórur Biskopstø Strøm, Oktay Baris, and Jens Sparsø. Scratchpad memories with ownership. In *2019 Design, Automation and Test in Europe Conference Exhibition (DATE)*, 2019.

[29] Bill Venners, Lex Spoon, and Martin Odersky. *Programming in Scala, 3rd Edition*. Artima Inc, 2016.

[30] Andrew Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2016.

[31] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. The RISC-V instruction set manual, volume I: Base user-level ISA. Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley, May 2011.

[32] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, Zuojun Li, Jiawei Lin, Tong Liu, Zhigang Liu, Jiazhan Tan, Huaqiang Wang, Huizhe Wang, Kaifan Wang, Chuanqi Zhang, Fawang Zhang, Linjuan Zhang, Zifei Zhang, Yangyang Zhao, Yaoyang Zhou, Yike Zhou, Jiangrui Zou, Ye Cai, Dandan Huan, Zusong Li, Jiye Zhao, Zihao Chen, Wei He, Qiyuan Quan, Xingwu Liu, Sa Wang, Kan Shi, Ninghui Sun, and Yungang Bao. Towards Developing High Performance RISC-V Processors Using Agile Methodology. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1178–1199, 2022.

[33] Jerry Zhao, Animesh Agrawal, Borivoje Nikolic, and Krste Asanović. Constellation: An open-source SoC-capable NoC generator. In *2022 15th IEEE/ACM International Workshop on Network on Chip Architectures (NoCArc)*, pages 1–7, 2022.

[34] Michael Zimmer. *Predictable Processors for Mixed-Criticality Systems and Precision-Timed I/O*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2015.

Index